DUDLEY K[illegible] LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

Approved for public release; distribution is unlimited

Turtle Graphics Implementation Using a Graphical Dataflow Programming Approach

by
Robert Steven Lovejoy
Lieutenant, United States Navy
B.S., Bradley University, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1992

UNCLASSIFIED

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
. DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
. NAME OF PERFORMING ORGANIZATION omputer Science Dept. aval Postgraduate School / 6b. OFFICE SYMBOL (if applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
. ADDRESS (City, State, and ZIP Code) onterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNI ACCESSIO

. TITLE (Include Security Classification)
URTLE GRAPHICS IMPLEMENTATION USING A GRAPHICAL DATAFLOW PROGRAMMING APPROACH

. PERSONAL AUTHOR(S)
Lovejoy, Robert Steven

a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
aster's Thesis	FROM _____ TO: _____	1992, September	194

. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the offic policy or position of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	OBJECT ORIENTED PROGRAMMING, TURTLE GRAPHICS, VISUAL DATAFLOW PROGRAMMING

. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis expands the concepts of object-oriented programming to implement a visual dataflow programı nguage. The main thrust of this research is to develop a functional prototype language, based upon the Turtle Gr s tool provided by LOGO programming language, for children to develop both their problem solving skills, as their general programming skills.

The language developed for this thesis was implemented in the object-oriented, dataflow programming lang ograph. The dataflow paradigm was emulated in order to provide a more intuitive, easy to learn programmin ronment for children to use. Additionally, Prograph was chosen because it provides the necessary base class sily implement an interactive user interface, and it provides the necessary primitive operations for all gra awing routines.

This thesis demonstrates a prototype for a potential visual programming language that can be used at all leve lucation to teach problem solving, higher-order thinking skills, mathematical concepts, and the fundamenta mputer science.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
a. NAME OF RESPONSIBLE INDIVIDUAL Thomas Wu, Prof., Computer Science Dept, NPS	22b. TELEPHONE (Include Area Code) (408) 646-2174 / 22c. OFFICE SYMBOL CS/Wu

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGI UNCLASSIFIED

ABSTRACT

This thesis expands the concepts of object-oriented programming to implement a visual dataflow programming language. The main thrust of this research is to develop a functional prototype language, based upon the Turtle Graphics tool provided by LOGO programming language, for children to develop both their problem solving skills as well as their general programming skills.

The language developed for this thesis was implemented in the object-oriented, dataflow programming language Prograph. The dataflow paradigm was emulated in order to provide a more intuitive, easy to learn programming environment for children to use. Additionally, Prograph was chosen because it provides the necessary base classes to easily implement an interactive user interface and it provides the necessary primitive operations for all graphics drawing routines

This thesis demonstrates a prototype for a potential visual programming language that can be used at all levels of education to teach problem solving, higher-order thinking skills, mathematical concepts, and the fundamentals of computer science.

T083
L83333
c.1

Table Of Contents

I. INTRODUCTION 1
II. SURVEY OF THE LITERATURE 3
A. OBJECT-ORIENTED PROGRAMMING 3
1. Classes/Objects and Related Variables and Methods 4
2. Inheritance 5
3. Encapsulation 7
4. Polymorphism 8
B. TURTLE GRAPHICS PROGRAMMING LANGUAGE 9
1. Turtle Graphics Origin 9
2. Overall Educational Benefit 10
a. Case Studies 10
b. Problem Solving Skills 11
c. Specific Curriculum Benefits 12
3. The Language 13
a. Turtle Space 13
b. Making Shapes 14
c. Making Procedures 14
d. Generalizing Procedures 15
C. PROGRAPH: A Visual Dataflow Program Style 16
1. Visual Systems-Iconic Based 17
a. Classes 17
b. Attributes 18
c. Methods 19
2. Visual Systems-Dataflow Based 20
a. Message Passing/Invoking a Method 22
b. Control Structures 23
III. DATAFLOW TURTLE GRAPHICS 28
A. LANGUAGE EVOLUTION 28
B. WHY VISUAL PROGRAMMING 29

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

1. Dual Brain Theory 29
2. A Need For a New Programming Style 30
C. WHY DATAFLOW PROGRAMMING 30
1. Executable versus Non-Executable Diagrams 31
2. Dataflow Functionality 31
D. ICONIC LANGUAGES 32
1. Iconic Guidelines 32
E. TURTLE GRAPHICS DESIGN AND IMPLEMENTATION 33
1. Developing Turtle Class/Objects and Methods 34
a. Class Hierarchy 34
b. Turtle/pTurtle Class Definitions 35
c. User Interface Design and Implementation: First Phase 36
2. Integration of Turtle Code and Dataflow Programming Code 39
a. Class Hierarchy 39
b. DFObject and Descendents 40
c. User Interface Design and Implementation: Second Phase 41
d. Program Objects: Icon-Description and Functionality 44
IV. PROBLEM SOLVING WITH DATAFLOW TURTLE GRAPHICS 48
A. GENERAL DISCUSSION 48
B. PROBLEM STATEMENT 48
C. DEVELOPING A SOLUTION 48
1. DFTG's Object-Oriented Approach to Problem Solving 48
2. "Man-Project" Problem Reduction 49
a. Create Turtles 49
b. Creating the Head 49
c. Create a Face 50
d. Create a body 50
e. Create a bowtie 51
f. Create legs 53

g. Create arms 54
h. Final Code Encapsulation 54
V. SUMMARY, CONCLUSIONS, & SUGGESTIONS FOR FUTURE RESEARCH 58
A. SUMMARY 58
B. CONCLUSIONS 59
C. SUGGESTIONS FOR FUTURE RESEARCH 59
1. Completion of "user-defined Turtle command" functionality 60
2. Completion of "user-help" functionality 60
3. Expand language control constructs 60
4. Fully implement Error detection/correction capabilities 60
5. Incorporate a programming pallet of available commands 61
6. Implement additional Turtle functionality 61
7. Perform statistical studies of user effectiveness 61
APPENDIX A - USER COMMAND/METHOD DEFINITIONS 62
APPENDIX B - NEW TURTLE GRAPHICS - SOURCE CODE 66
LIST OF REFERENCES 180
BIBLIOGRAPHY 182
INITIAL DISTRIBUTION LIST 183

List of Figures

Figure 2.1 Superclass/Subclass Inheritance Hierarchy 6

Figure 2.2 Ship Example 7

Figure 2.3 Turtle Graphics commands for Square-process 14

Figure 2.4 Graphical Class Hierarchy 18

Figure 2.5 Application Attributes/Methods Windows 19

Figure 2.6 Case Window for Window/Close 21

Figure 2.7 Method Calling Formats 22

Figure 2.8 Case on Fail Control Structure 25

Figure 2.9 Case Success Control Structure 26

Figure 2.10 Synchro Control Structure 26

Figure 3.1 Class Hierarchy 34

Figure 3.2 Turtle Class Definition 35

Figure 3.3 pTurtle Class Definition 36

Figure 3.4 Graphics Display Window 37

Figure 3.5 Turtles Menu Option 38

Figure 3.6 Turtle Attribute Interface Window 39

Figure 3.7 Dataflow Turtle Graphics Class Hierarchy 40

Figure 3.8 Dataflow Programming Window 42

Figure 3.9 Revised Turtle Display Window 43

Figure 3.10 Turtle Graphics Help Window 44

Figure 3.11 Programming Object Icons 45

Figure 3.12 Stored Program Code for Star 46

Figure 3.13 User-Defined Operator Code 47

Figure 4.1 Head Solution 50
Figure 4.2 Face Solution 51
Figure 4.3 Head/Face Integration 52
Figure 4.4 Body Solution 52
Figure 4.5 Bowtie Solution 53
Figure 4.6 Integrated Body/Bowtie Solution 53
Figure 4.7 Head/Body Integrated Solution 54
Figure 4.8 Legs Solution 55
Figure 4.9 Head/Body/Legs Integration 55
Figure 4.10 Arms Solution 56
Figure 4.11 Complete Integration 56
Figure 4.12 Final Man Encapsulation 57

ACKNOWLEDGEMENTS

This thesis was made possible through the efforts of several people. First and foremost, thanks to my advisors Dr. Wu and Dr. Erickson. Had it not been for their challenging questions and patient guidance, the completion of this research would have not been possible.

Special thanks goes to John Daley, LCDR, USN, a military instructor at Naval Postgraduate School. Much valuable research information was gained with his guidance to related Turtle Graphics material.

Lastly, for the most important people in my life, my family, without whom I certainly would not be where I am today. Thanks for all the encouragement, love, and support, especially from my wife Vikki. I only wish that I had been able to spend more time with you and the kids. And finally a special thank you to my children, Michael, Brittany, and Andrew for making me smile even when my mind was pre-occupied.

I. INTRODUCTION

The purpose of this thesis is to expand upon the graphics portion of *LOGO*[1] programming language. Much research has been conducted in the area of Turtle Graphics languages, however, they are text-based implementations requiring a relatively high degree of sophistication with text and language constructs [CIL86]. The intent of this research is to design and implement *LOGO's* turtle metaphor into a Turtle Graphics Dataflow Programming Language. The major areas of concern in this thesis are Object-Oriented Program Design, Turtle Graphics Programming Language, and Visual Dataflow Programming.

Dataflow Turtle Graphics (DFTG) has been developed as a language for children to develop their problem solving skills as well as basic programming concepts. It is a tool to teach the process of learning and thinking. DFTG is a visual programming language which supports the execution of dataflow programs. It was implemented with an object oriented design using Prograph[2] [TGS88a, TGS88b, TGS91], an object-oriented programming language (OOPL) available on the Apple Macintosh[3]. This language was chosen because it provided the necessary base classes for interface design, as well as the primitive operations for all graphics drawing functions. Prograph also handles list processing and manipulation of non-conventional objects (i.e., pictures, sounds, etc.) very easily, which is important to the languages' continued expansion.

The main thrust of this thesis was to implement a prototype language, DFTG, by combining the concepts of Turtle Graphics Programming with Visual Dataflow

---

1. Armedeus is a visual, object-oriented database, thesis developed by several students under advisement by C. Thomas Wu, Prof., Computer Science Department, Naval Postgraduate School, Monterey, Ca.

2. Prograph is a trademark of The Gunakara Sun Systems, Ltd.

3. Apple and Macintosh are registered trademarks of Apple Computers, Inc.

Programming. The essential theme is to remove the short comings of text-based programming, and to provide a more intuitive, easy-to-learn environment for dataflow programming.

The remainder of this thesis is organized as follows. Chapter II is a survey of the literature that forms the background for this research. It lays the groundwork for future discussion in this thesis, and provides an overview of the main topics of this thesis: Dataflow Programming, Turtle Graphics, and Object-Oriented Program Design. Chapter III presents a detailed description of the design and implementation of Dataflow Turtle Graphics. Chapter IV provides a step-by-step dataflow turtle graphics solution for a particular programming project. Chapter V provides a summary, conclusions, and suggestions for future research. Appendix A provides the definitions for all predefined user commands. Appendix B provides the source code for the implementation of this thesis.

II. SURVEY OF THE LITERATURE

This chapter deals with three major topics: Object-Oriented Programming (OOP), Visual Dataflow Programming Languages, and Turtle Graphics Programming Language. Basic terminology and concepts are discussed in this chapter. Prior knowledge of these areas is not required for understanding the intent of the research. This chapter is intended to serve as an introduction to these three topics, laying the groundwork for the rest of this thesis.

A. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming is a relatively new area of programming whose origin has been attributed to the programming languages Simula and Smalltalk [Booc91]. Although OOP seems to be on the rise in programming and program design today, there is clearly no single standard to abide by in order to be labeled an object oriented language. Clearly, the motivation for all such languages is to provide faster development, reliable and quality products, and easier maintenance and extension.

Object oriented programming languages provide four main features to achieve their programming goals: abstraction, encapsulation, inheritance, and polymorphism. Abstraction supports code reusability, shareability, and allows for integration. Encapsulation supports code reliability, extensibility, and also allows for integration. Inheritance supports code reusability, shareability, and extensibility. Lastly, polymorphism supports code extensibility, and shareability.

Creating complex applications using an object-oriented programming language (OOPL) can be simpler than designing the same program using a more conventional procedural language. This is because OO design more closely mirrors the real world entities being modeled. Additionally, the full benefit of OOP can only be realized if

encapsulation is maximized during the design process. As program complexity increases, so to does the benefits of an object-oriented design. The most effective tool for dealing with this complexity is abstraction. The most common form of abstraction by which complexity is managed is encapsulation.

The following are generally considered to be the fundamental characteristics of all object-oriented programming languages: class/object and associated variables and methods, inheritance, and polymorphism. The very basics of each of these concepts follows:

1. Classes/Objects and Related Variables and Methods

A ***class*** simply defines a mold or template from which all ***objects*** or instances are cast. It specifies a particular set of characteristics used for defining objects of that class or, more formally, "a set of objects that share a common structure and a common behavior" [Booc91; page 93]. Once a class is defined it does not change. In object-oriented programming an ***object*** is an abstraction of a real-world entity. They are specific instances of a class having their own set of self-contained variables and behaviors by the use of methods. Each instance of a class can have a unique set of values assigned to its variables or attributes, which may or may not change throughout the execution of the program. Objects are intended to encapsulate both data and behavior.

Consider the following example of class/object. Declare **Ship** as the class or template of the objects desired. The class description serves as an abstract description of related objects and how they interact with each other and the outside world. As stated, it is helpful to think of a class as the general description of a real-world entity. This particular class will describe some of the common characteristics of all ships in general. A specific **Ship**, such as USS JARRETT, is an object (instance) of this class. All objects of class **Ship** share the same structure and

behavioral aspects. It is this basic structure and behavior aspects that defines the class. All instances of the class **Ship** will have its own set of instance variables. Each individual ship will have values for these instance variables specific to that ship. Furthermore, instance variables share only their name with other instances of that class. Their values are independent of each other. Class variables, on the other hand, remain identical for all instances of that class, in name and value.

A ***Method*** is a procedure or function associated to the instance of a class that defines their behavior. Methods are invoked by passing messages to objects. The object will respond appropriately if there exists a method, of the same name, within its own set of methods. This set of methods defines the objects behaviors. Methods are generally the only means for other objects to access the variables of a specific instance. Methods are also broken into at least two groups, instance or object and class methods. In **Ship** class, *create_ship* is an example of a class method to be used when adding a new ship is required. An instance method may be *get_ship_type*, and would be directed at a specific instance of a class.

2. Inheritance

Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses [Booc91]. Traditional procedural languages, such as Ada, do not support this object-oriented feature. Inheritance is a means for programmers to construct *reusable* objects so they can produce programs in a relatively short time through code sharing. New classes can be easily defined based on existing classes. The new class is referred to as a ***subclass*** of the existing ***superclass***. Figure 2.1 represents a general

class hierarchy. Classes Y and Z represent subclasses of Class X with arrows indicating the direction of inheritance. Class X may also be referred to as the superclass of classes Y and Z. Y and Z inherits all of the attributes and methods of Class X. Additionally, subclasses may add more variables and methods as required to define that class. Subclass method names may or amy not be unique. The effect of using the same method name for a subclass will simply overshadow or change the behavior of messages sent to instances of this class. All messages received by an object will first check the methods of that class for a match and will proceed up the inheritance chain until a match is found.

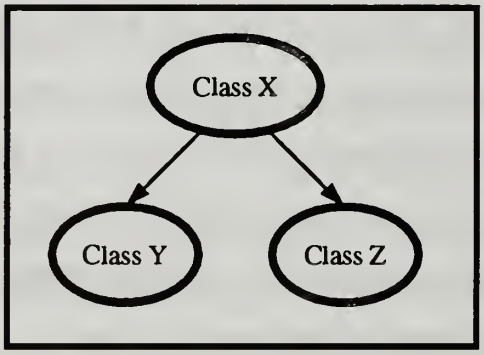


Figure 2.1 Superclass/Subclass Inheritance Hierarchy

There are two types of inheritance, single inheritance and multiple inheritance. ***Single inheritance*** is defined as a class that can have only one superclass. ***Multiple inheritance*** is defined as a class which inherits form more than one superclasses. The specific considerations of single vs. multiple inheritance will not be discussed in this research.

Figure 2.2 shows how a typical class hierarchy might appear for our **Ship** class. **Ship** would be considered the superclass of **Supply**, **Combatant**, **Auxiliary**, ect. **Ship** might have the attributes *name, hull, homeport, draft,* ect. Each of these attributes are common in all ship types and should be defined in the class template.

The subclasses would each inherit all of the superclass attributes and methods, but would likely require some augmenting to satisfy specific needs of each type of ship.

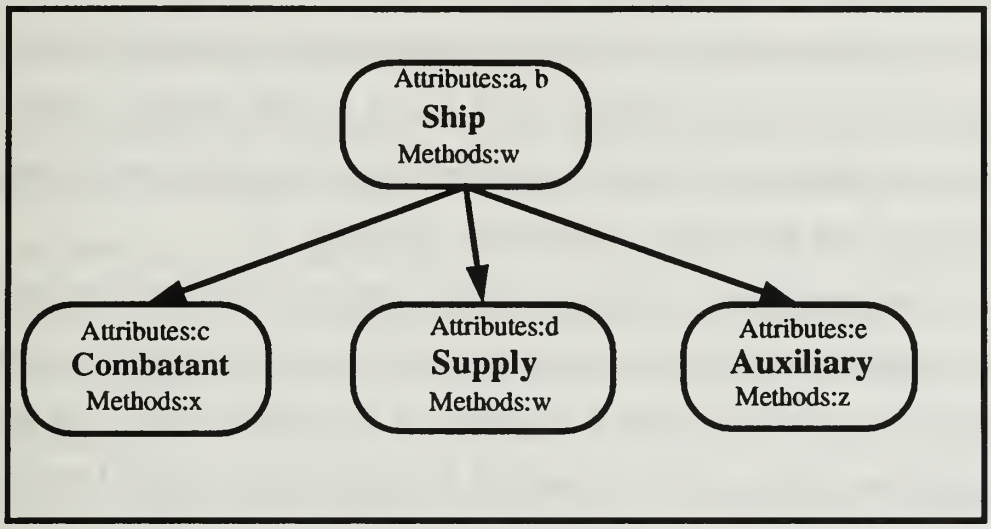


Figure 2.2 Ship Example

In the Ship example, objects of all subclasses would inherit attributes *a* and *b* from **Ship** class, as well as the behavior to respond to method *w*. Each subclass has also augmented its' attribute list with its' own specific attributes. Additionally, subclass **Supply** has overshadowed the superclass method, *w*, thereby changing the response behavior if it should receive a message *w*.

3. Encapsulation

Encapsulation is a means of storing an objects attributes and methods in a kind of black box. It can be defined as "the process of hiding all of the details of an object that do not contribute to its essential characteristics" [Booc91; page 46]. It is commonly referred to as *information hiding*, and is the most effective means of managing the complexity of a program. Programming in an object-oriented language, however, does not ensure that the complexity of an application will be well encapsulated. Applying good programming techniques can improve encapsulation, however the full benefit of object-oriented programming can only be

realized if encapsulation is a main goal during the design process. In OOP, a properly encapsulated program will provide a more extendable, easily modifiable, and integratable application. The black box approach ensures that the user does not need to know the internal details about how a specific method works, or what attributes the object has. The user needs only to be aware of the name of the message to be passed, and what will be returned by the object.

Encapsulation provides the means by which a developer can have several teams working on different object specifications, to be tested separately for correctness, and then integrated together for the final product. The modularity concept of producing code was essential for developing the final Dataflow Turtle Graphics Language. It also allows for the code to be improved/modified without affecting how end users access the object.

4. Polymorphism

Polymorphism along with inheritance form the very basis of object-oriented philosophy. **Polymorphism** is a phenomenon that occurs when the same message is sent to different objects. Each object responds with a method appropriate to its class.

Polymorphism allows programmers to add methods with the same name to classes that share some commonality and therefore use the same name to denote the specific function. Consider a graphics application in which a window is to drawn with various different objects. Appropriate responses would result with the same **Draw** message being sent to each object individually.

Polymorphism, used correctly, does away with elaborate control structures to handle all possible scenarios. Without polymorphism, the **Draw** method example would have to modified each time a new kind of object was added. With polymorphism, however, no changes are required to be made to the existing code.

Thus, polymorphism, facilitates code extensibility and modifiability in less time and with less errors.

B. TURTLE GRAPHICS PROGRAMMING LANGUAGE

Turtle Graphics can be thought of as a programming language for learning. It is a language that encourages students to explore, learn, and think. It provides all the tools required to create programs of varying degree of difficulty. Through immediate, visual, and non-judgmental feedback, the student feels in complete control of the graphics program, and thus is motivated to continue on the problem-solving journey. In a creative and helpful environment, Turtle Graphics turns mistakes into opportunities for exploration and new creation. In all, Turtle Graphics helps students with personal development, attitudes toward learning, depth of understanding, and other long-term benefits.

1. Turtle Graphics Origin

Professor Seymour Papert first introduced Turtle Graphics with the development of the programming language LOGO at MIT in 1967. The initial intent was to develope a computer language that would be both suitable for children, yet powerful enough for the professional programmer. The name LOGO was chosen to suggest the fact that it is primarily symbolic and only secondary quantitative [Pape80,pg210]. The Turtle is an example of a constructed computational "object-to-think-with." The principal role of the Turtle is to serve as a model for other objects, yet to be invented. The Turtle is simply a computer-controlled cybernetic animal. It exists within the cognitive LOGO environment, LOGO being the computer language in which communication with the Turtle takes place. The Turtle serves no other purpose than of being good to program and good to think with. It is generally assumed that the more powerful a programming language is, the harder it is to learn. LOGO is based on the concept of easier learning, by relating a turtle to

an object used to think with. Turtle Graphics provides a straightforward meaning to attach to each individual procedure, namely, a picture. The basic foundation for Turtle Graphics lies with the idea that specific problems of interest to the novice can be tackled by simple programs.

2. Overall Educational Benefit

LOGO involves more than just manipulating a turtle object or using mathematics. Its essence involves thinking about processes and about how you are doing what you are doing. In some cases of educational development, the process of creating a product is more important than the final product. Indeed, it may be more interesting to look at how a design was created than to look at the design itself.

a. Case Studies

Much research has been conducted on the relative benefits of learning LOGO through Turtle Graphics. Although there are some studies with mixed or inconclusive results, one conclusion is clear: The teacher is critical to the students' success. Some of LOGO's supporting case studies follow:

In one study, 45 third grade students were split into one of three groups, of which two used LOGO and the third used an array of other "problem solving" software. One LOGO group used problem solving strategies to solve graphics problems, while the other used LOGO to solve geometry problems, The same teachers using the same instructional methods, rotated through the groups. The results showed no difference in the groups' general problem solving ability. However, those in the LOGO groups "planned more effectively" and "represented the planning task differently" from the non-LOGO group. In both LOGO groups, there was an increased understanding of geometry [LGL88].

In another study of four seventh grade mathematics classes, two classes substituted one period per week of LOGO activities for traditional geometry

instruction. In pre-tests. the non-LOGO classes scored higher. However, at years end, on a 60-item test on applications of angle estimation, the LOGO classes improved 22% versus the non-LOGO classes' 13%. Significant differences between the two groups were found in all six areas of the post-test[Fraz87].

This study investigated the learning of fractions in the fourth grade, and how LOGO affects the students' understanding. All the fourth graders received the same instruction on fractions. The control groups learned to program in LOGO, but with no attempt to relate LOGO to their study of fractions. The test group was asked to design software about fractions that they could present to third graders. At the end of the study, the test group performed better than the control groups in knowledge about fractions (as measured by standardized test) and in LOGO programming ability. They also persevered in solving problems [Hare88].

This final study involved 48 children in grades 1 and 3, who were randomly assigned to 28 sessions of either LOGO or drill and practice work. They worked in pairs and were observed in terms of social interaction and problem solving activity. When significant differences were found between the groups, they favored the LOGO group. These differences occurred in three of seven categories of social behavior defined for the study: resolution of conflict, self-direction and rule determination. This study supports the use of LOGO as a means of encouraging desirable social interactions that are likely to lead to subsequent problem solving behaviors [CN88].

b. Problem Solving Skills

If children learn nothing else, in their early years of development, they must learn some sort of general problem solving skills. For any project, their must be an identifiable and attainable goal to reach. Given an idea for a project, consider the following problem solving scenario.

Initially spend some time just thinking about how to approach the problem, then experiment with some ideas. It is then necessary to break the problem into small, manageable chunks, and solve the individual little problems. If one way doesn't work, then think of another. The ability of learning to look at a problem from different approaches is the result of continuing to try new ideas and observing their results. Once a better way of doing something is observed, the idea is modified and tested. This becomes a cyclic approach until each little chunk is acceptable. The smaller programs are then combined into the larger solution. This forms the basis of a solution that is clear, precise, and with no ambiguity. The entire solution becomes a sequential organization of fluid ideas that remain easy to understand and return to.

As researchers try to assess LOGO's ability to improve problem solving, they face the same difficulties as they have for years: problem solving in any environment is extremely difficult to evaluate [YM90].

c. *Specific Curriculum Benefits*

LOGO's Turtle Graphics provides the necessary structured programming environment, that enhances one's ability to creatively learn. The underlying nature of Turtle Graphics can be beneficial to multiple curriculums of education.

(1) *Mathematics*: This area is probably seen as most influenced by Turtle Graphics. Estimation is introduced by working with distances and angles. Polygons use REPEAT to create regular shapes. Number relationships are investigated using perimeter and area. Symmetry is necessary when drawing points and lines. Learning a coordinate system is required for plotting points and graphing lines. Geometry is reinforced through drawing and measuring lines and angles.

(2) *Programming*: Proper techniques form the basis for writing structured programs. Program design is accomplished through breaking down a

problem into smaller tasks. Flow of control is accomplished with branching and conditionals. Variables and recursion exemplify the power of the language.

(3) *Social Studies*: A sense of direction is accomplished through translation of the turtle's heading into compass points. Cartography can be introduced by making maps with Turtle Graphics. The concept of learning a foreign language is apparent, as a result of creating and using foreign language primitive and procedure names.

3. The Language

Although LOGO language is a completely, full-featured programming language, it is only necessary to concentrate on the Turtle Graphics portion for the extent of this research. The intent of this section is not to teach how to program in LOGO, with the use of Turtle Graphics, but simply to provide an introduction to the Turtle Graphics concepts and programming environment.

a. Turtle Space

Turtle space is defined by the dimensions of the screen on which the graphics is to be displayed. Screen dimensions are generally stated in vertical (the y-axis direction) and horizontal (the x-axis direction) measurements. It is imperative to pinpoint the origin of the screen where x,y =0.

The turtle can be moved about the screen using cartesian x-y coordinates or turtle coordinates. Cartesian commands send the turtle to a specific x-y position on the screen, without regard to the turtles current position.

In the turtle reference, all commands refer to the turtle's current position, not its final position. The turtle is moved forward, backward, turned left or right in relation to where it is now.

In the cartesian system, the destination is the important thing; in the turtle reference system, it's the trip [Clay88].

b. Making Shapes

Turtle Graphics provides the necessary tool to draw graphics using turtle reference commands. Consider using a turtle draw a square. The following steps would be required, using the turtle metaphor to think through the process, to complete the task.

1. "OK turtle, go forward 50 steps and turn right by 90 degrees. That completes the left side of the box."
2. "Now, go forward another 50 steps and turn right by 90 degrees. That completes the top of the box."
3. "Go forward yet another 50 steps and turn right, again by 90 degrees. That completes the right side of the box."
4. "Go forward another 50 steps and turn right 90 degrees. That completes the bottom edge of the box."

That completes the process to make a square. Figure 2.3 shows the results of the turtle task, as well as two equivalent LOGO command translations. The **Repeat** command shows the power of iteration in LOGO.

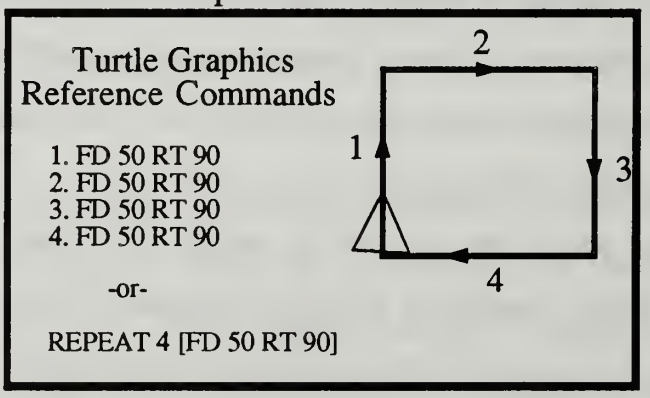


Figure 2.3 Turtle Graphics commands for Square-process

c. Making Procedures

It is sometimes convenient and necessary to be able to define a particular process for further use. A series of LOGO commands may be grouped

together under a single name by writing a LOGO procedure. The name of the procedure is a shorthand for all commands included in it, a form of encapsulation. Typing the name of the procedure tells LOGO to automatically execute each line of procedure in turn, just as if you had typed them, one after another, on the keyboard.

In the Square-process example, it is possible to define a square procedure, for further reference, in the following manner:

```
TO SQUARE50
  REPEAT 4 [FD 50 RT 90]
END
```

LOGO will add SQUARE50 to all its other commands. Each time SQUARE50 is typed, the turtle will draw a square of size 50. The figure will be drawn at the turtle's current position on the screen.

d. Generalizing Procedures

Generalizing a procedure can add to the power of the command by giving the user more control and flexibility. Consider the SQUARE50 procedure. This procedure could be edited each time a square of new dimensions is required, or many SQUARE-like procedures could be defined, however this is not very efficient. After all, LOGO itself does not have multiple 'FD' commands for every possible length of a line drawn.

Arguments must accompany LOGO commands. Arguments provide the command with the necessary missing information to complete its task. For example, the line-drawing command, FD, must be accompanied by an integer argument so that LOGO knows the correct length of the line to be drawn.

Consider the SQUARE50 procedure. The value of the argument will tell the square procedure how long each side is to be drawn. Changing the value of

this argument will result in boxes of different sizes. The new generalized SQUARE procedure would look as follows:

```
TO SQUARE
  REPEAT 4 [FD:EDGE RT 90]
END
```

The preceding concepts and examples provide the basic foundation for the creation of Turtle Graphics as a programming tool used in LOGO. It is precisely these basic foundations that provide a point of departure for the implementation of Dataflow Turtle Graphics presented in Chapter IV.

C. PROGRAPH: A Visual Dataflow Program Style

Prograph [TGS88a, TGS88b, TGS91] is stated to be a "very high-level, pictorial object-oriented programming environment", which integrates several areas of computer science. Additionally, Prograph supports an object-oriented application building toolkit. This section explores Prograph's visual programming environment including the use of dataflow diagrams for method definitions, and it's use of icons as programming language constructs. As a complete language, Prograph satisfies a wide range of different programming requirements.

Prograph has been characterized as a hybrid OOP language, since it supports primitive language types such as integer, boolean, character, etc. A pure OOP language has no primitive language types; everything is an object [Booc91]. Additionally, Prograph supports a feature which incorporates the use of universal methods, thus adding to its hybrid likeness. These methods do not belong to any particular class, but can be called from any method in any class [TGS88b].

The intent of this section is not to teach how to program in Prograph, but only to provide a basic understanding the Prograph language, and its programming environment. Several examples are taken from actual code provided by Prograph lan-

guage. Specific features are highlighted and discussed to provide an understanding of how programs are written in Prograph.

1. Visual Systems-Iconic Based

There is no clear-cut definition as to what is meant by the term "visual programming", however, in general, it refers to the use of graphical representations in the process of programming. This programming style is an extreme departure from traditional programming and is not dependent on linguistic ability or limited by the user's knowledge of verbal syntax. Visual programming involves nonverbal, visual information that is recognized and understood in a single, simultaneous process.

Prograph is a fully visual development environment, as well as a fully specified icon-based language. In contrast to text-based systems, icon-based systems use pictures as programming language constructs, that is, executable graphics. Prograph supports a highly visual programming system which has multiple windows for viewing program execution states, visual syntax editors for designing program data structures, and graphical expressions in the windows themselves. Figure 2.4 is a typical example of the visual nature of Prograph. It shows a graphical representation of the hierarchy of base classes provided by the language.

a. Classes

Figure 2.4 shows the classes window for the base classes provided by Prograph. All applications start out with these minimum template classes. Each Prograph class is represented by a hexagonal icon displayed in the Classes window. All class hierarchies for the program are displayed in the classes window. There can be multiple class hierarchies, as required by the application. The lines connecting individual classes within the hierarchy represent the inheritance links between various classes. Prograph supports an upward inheritance.

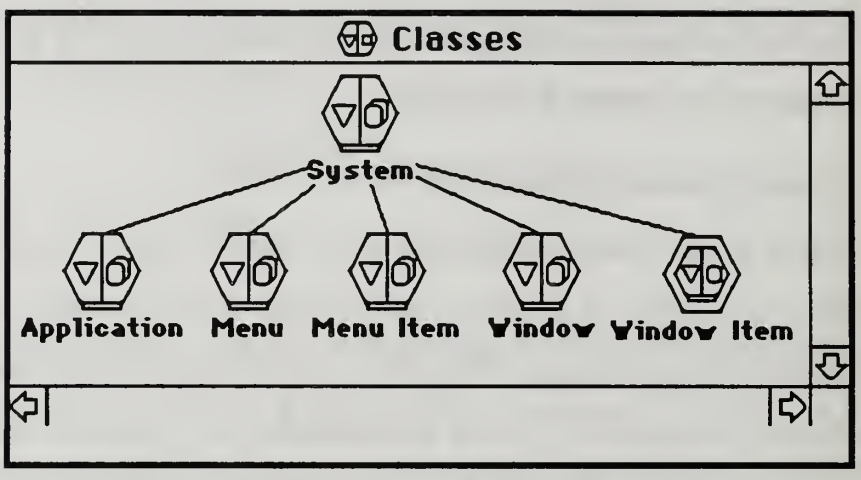


Figure 2.4 Graphical Class Hierarchy

The class icon itself is divided into a left and right half. The triangle on the left-half of the icon represents the attributes of the class while the stacked rectangles on the right-half represents the methods. Double-clicking on the left half opens the attributes window for the particular class. Similarly, double-clicking the right half opens the methods window for the class.

New classes are created, and will appear, by clicking inside of the classes window. The new class is then given a unique name and is defined by adding the appropriate attributes and methods. Attributes and methods are also created by clicking in their respective windows.

b. Attributes

Figure 2.5 shows the results of double-clicking on the left and right halves of the class icon. Class attributes are represented by the hexagon shaped icons while instance attributes, below the gray line, are represented by inverted triangles. Inherited attributes have a downward pointing arrow inside of the triangle icon. Figure 2.5 shows both inherited attributes, and local attributes. Local attributes are not inherited and do not have the downward pointing arrow. Attributes can be

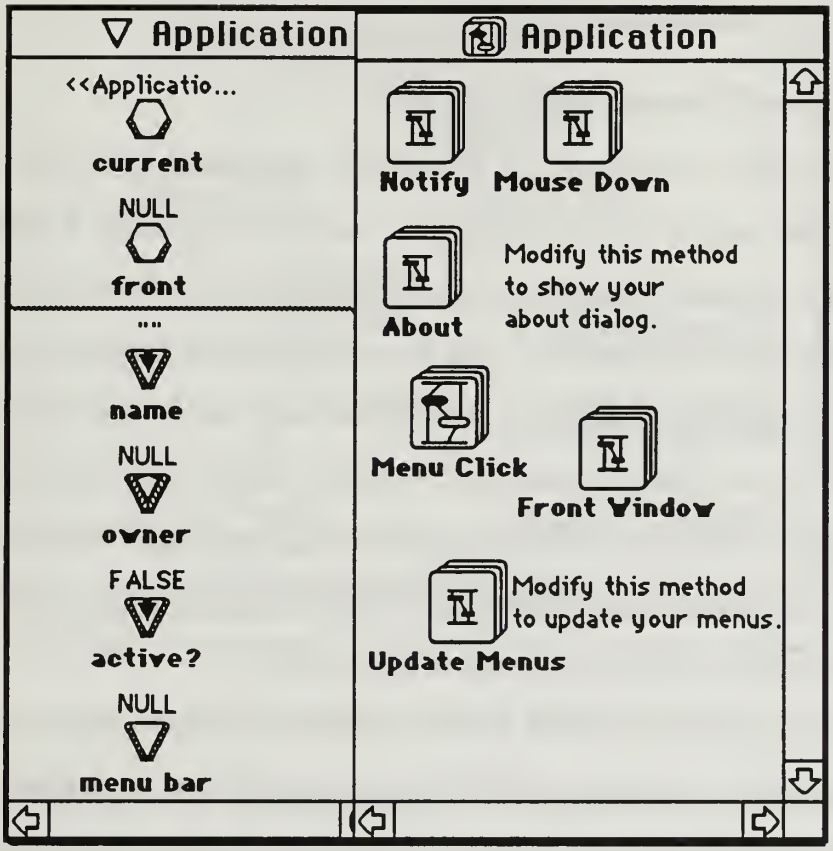


Figure 2.5 ***Application*** **Attributes/Methods Windows**

assigned initial values by double-clicking on the icon and changing the value in the attribute editor. Attributes can also be more than simple data types, they can be instances of other classes; this is a means of representing a composite object in Prograph.

c. Methods

Figure 2.5 also shows the Application-Class methods window. As seen, methods are represented by an icon that contains a small dataflow diagram. Additionally, their is a special type of method known as an instance generator, not shown. Its has an icon that is represented by the symbols <<>> in a hexagonal shape. This instance generator method may be invoked whenever an instance of that class

is created. This method also overshadows, by redefining inherited attributes or methods, the instance primitive.

2. Visual Systems-Dataflow Based

Dataflow programming potentially represents a means for efficiently exploiting the concurrency of computing on a very large scale. A dataflow language is any language either based entirely on the notion of data flowing from one function to another or directly supporting such flowing of data. In Prograph, active data flows through the program, activating each instruction as soon as all of its required input data have arrived. These instructions can be anything from a simple system-supplied primitive, to a call to an arbitrarily complex user or system defined method. While Prograph is inherently concurrent due to its dataflow design, the Macintosh is a single-processor, and therefore sequential, machine.

Figure 2.6 shows the result of double-clicking on a method icon. The lettering of the operators was added for clarity of discussion. Additionally, operators D,E, and F were added to support the review of various Prograph operators. This case window provides the dataflow programming interface window for defining the actual behavior of the method. Undefined methods will open with only an input bar along the top, and output bar along the bottom. Two terms critical to the dataflow paradigm are terminal and root. Terminals represent the input objects that allow data to flow into a method, while root represents the output object from which data flows out of a method. Their icons are small circles attached to the top and bottom of methods. They are uniquely numbered from left to right.

In the Window/Close method example, there are several types of Prograph operators. Operator A represents a *get-operation,* and will result in retrieving the labeled attribute value from the object flowing into it. Operator B represents a *set-operation*, which results in setting the labeled attribute value of the input object on

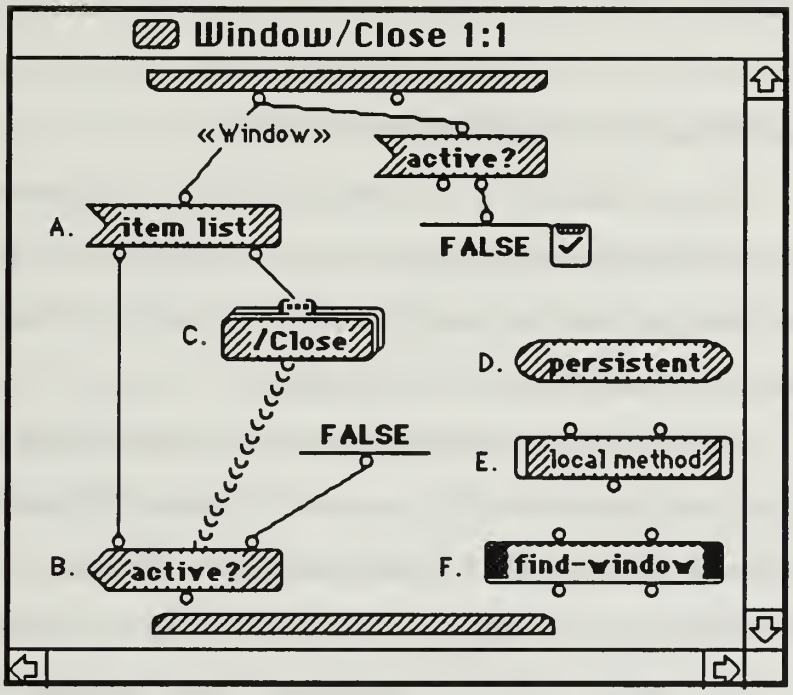


Figure 2.6 Case Window for Window/Close

terminal-1 to the value of terminal-2. Operator C represents one way of making a call to another method. The various ways to make method calls will be discussed later. Operator D represents a persistent operation. Prograph is one of the few OOPLs that supports persistent objects. Persistents are defined as data or objects that exists from one execution of a program to another. They are created and displayed in a Persistents window that is separate from the Classes window. Persistents are created in the same way as a class or method, and can be double-clicked to display their values. Persistents allow the user to manipulate objects and store them within the program so that they can be used later during the execution of the program, or recalled during another execution of the program. Operator E represents a local method operation and exemplify the encapsulation concept for managing complex methods. It is only accessible within the containing method. Operator F represents

a typical primitive operation. Primitive are pre-defined methods provided by Prograph.

a. Message Passing/Invoking a Method

Message passing, or invoking a method, in Prograph is accomplished by creating an operation with the same name of the method being called. Prograph assigns the operation the correct arity, number of input terminals and output roots, based upon the arity of the method being called.

Methods may be invoked in several manners. Figure 2.7 shows four of the most common means of calling a method. They include: universal reference, explicit reference, data-determined, context-determined reference.

Method A represents a *universal reference*, where the format is "method". This is simply a call to a predefined, global method. Prograph will look for the method *draw* in its universal methods file.

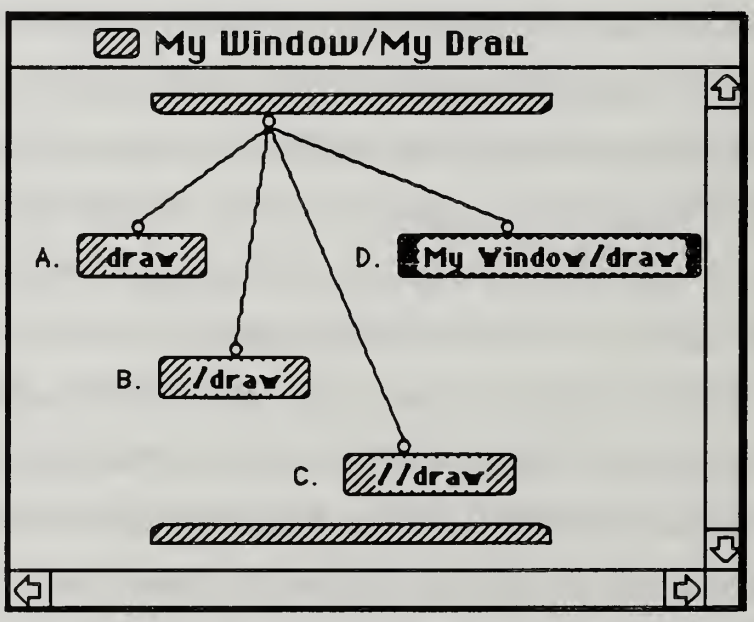


Figure 2.7 Method Calling Formats

Method B represents a *data-determined reference* and has a format of "/method." The class of the object entering on terminal-1, of the method, determines where Prograph will look for the proper method. Data-determined references exemplifies the concept of polymorphism.

Method C represents a *context-determined reference*, and is of the format "//method." This form of reference indicates that the named method is to be found in the same class as the current method that contains the method referencing operation. This is a means of sending a message to itself.

Method D represents an *explicit reference*, and is of the form "class/ method." Prograph attempts to find the specified method in the specified class. If the method is not found in the specified class, Prograph will use the inheritance link to check ancestor classes.

b. Control Structures

Control structures are essential features to any language. Prograph has an extensive set of control features. These structures are required to have positive control of the data as it flows through the program. These features are accessible through the *Controls* menu. A subset of Prograph's control structures will be discussed here-in-below.

Most programming languages provide a means of conditional program execution. In text-based languages, a particular syntax is used to structure such variations in program flow. Typical language constructs for conditional execution are: If <condition> Then <response> End or If <condition> Then <trueresponse> Else <falseresponse> End or While <condition> Do <this> End or typical Case statements.

The most basic Prograph conditional execution form is the **Next Case** annotation with a match operation or a conditional test based on one or more of the

available boolean primitives. Next Case annotation has two forms, Next Case on Failure, and Next Case on Success. These control structures are represented by a small box icon attached to the right of the operation. Next Case on Failure has an (X) enclosed in the box, while Next Case on Success has a (√) enclosed in a box.

Figure 2.8 shows the format of a typical Next Case on Failure. This structure attached to a boolean operation means, "if this test fails, go to the next case." The same structure attached to a match operation means, "if the value coming into this match operation is not equal to the constant value of this operation, go to the next case." The Next Case on Failure example is completely documented for further review and understanding of this control structure.

Figure 2.9 shows the format of a typical Next Case on Success. This structure attached to a boolean operation means, "if this test succeeds, go to the next case." The same structure attached to a match operation means, "if the value coming into this match operation is equal to the constant value of this operation, go to the next case." Additionally, the Next Case on Success example is completely documented for further review and understanding of this control structure.

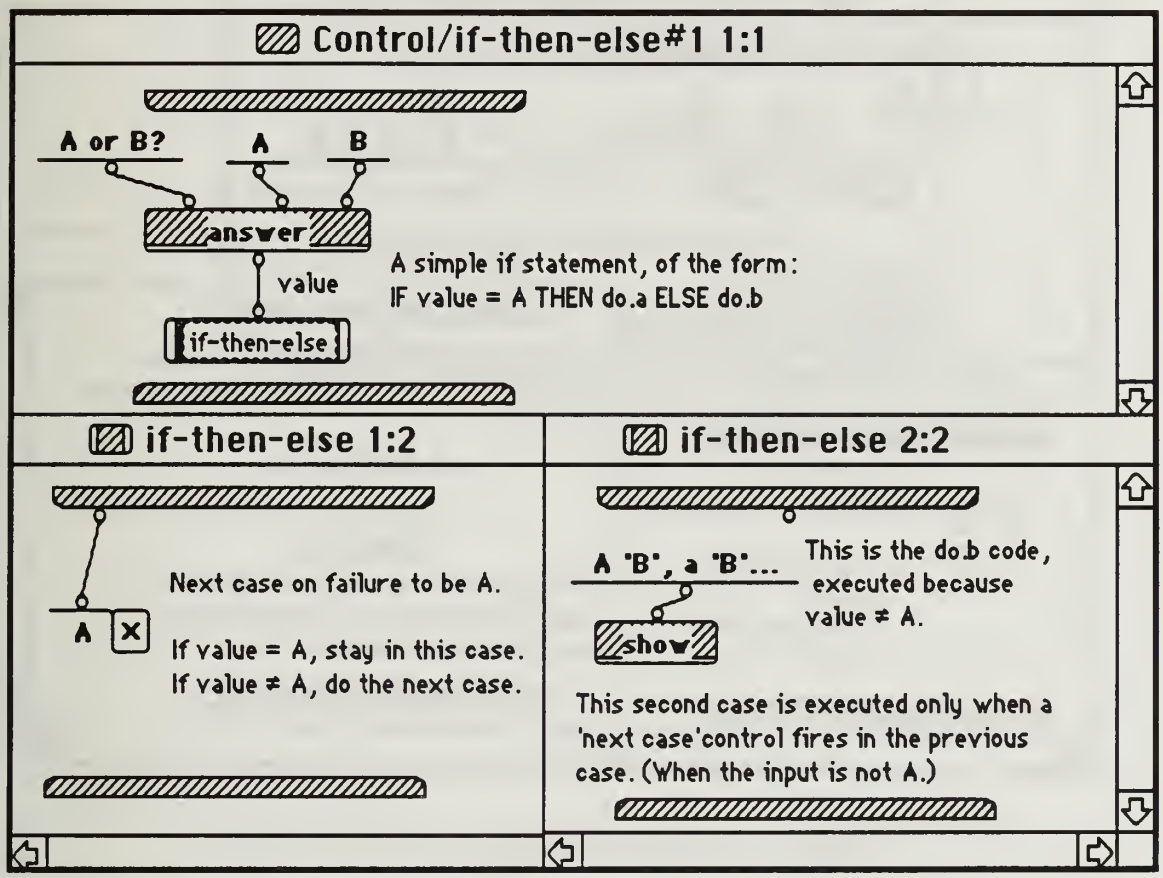


Figure 2.8 Case on Fail Control Structure

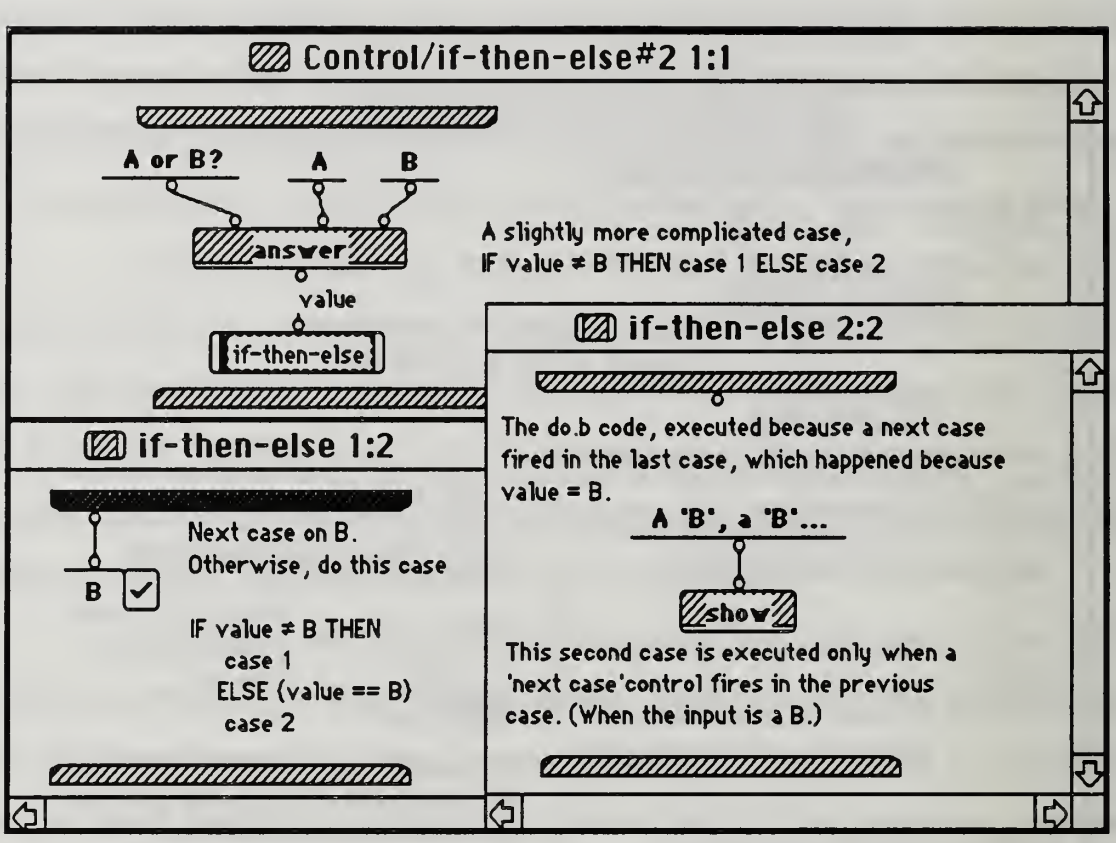


Figure 2.9 Case Success Control Structure

Since Prograph is inherently parallel, and an operations' execution is only dependent upon the availability of input data, control of the relative order of program execution can be very important. Figure 2.10 shows Prograph's *synchro* control structure. This enables the programmer to control the relative order of the execution of a program.

Additional Prograph control features include Continue, Finish, Fail, Inject, List, Loop, and Terminate structures, however, these won't be discussed herein.

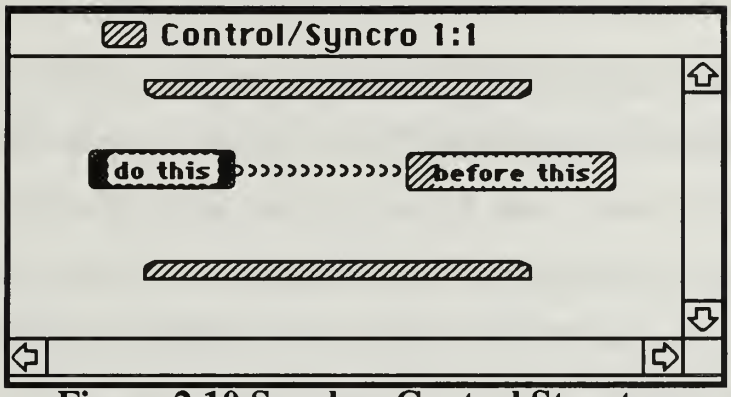


Figure 2.10 Synchro Control Structure

III. DATAFLOW TURTLE GRAPHICS

The previous chapters were presented to give the reader a basic understanding of Object-Oriented Programming, Turtle Graphics Programming, and the Visual Dataflow Programming with Prograph. This chapter presents the reasoning for implementing a Visual Dataflow Programming Turtle Graphics Language. Additionally, this chapter will explain the design and specifications considered in implementing DFTG.

A. LANGUAGE EVOLUTION

Programming Languages have evolved through multiple generations, over the last thirty years, from low level, to high level, to very high level, to ultra high level. Although there is no universal agreement on the division and definition of the different levels of languages, one characteristic stands out without much dispute: as the level goes up, fewer details are required from the user.

Another observation is that, with few exceptions, the tradition of linear representations persists from generation to generation. Instructions are given to the computer in a statement-by-statement manner. The structure of the programming languages remains one-dimensional and textual.

In contrast, visual programming represents a conceptually revolutionary departure from this tradition. Graphical representations and pictures have come into play in the programming process. This evolvement of the traditional programming language is stimulated by several premises.

1. Pictures are more powerful than words as a means of communication. They can convey more meaning in a more concise unit of expression.
2. Pictures aid understanding and remembering.
3. Pictures may provide an incentive for learning to program.
4. Pictures do not have language barriers. When properly designed, they are understood by people regardless of what language they speak.

Additionally, visual programming has gained momentum in the past few years because the falling cost of graphics-related hardware and software has made it feasible to use pictures as a means of communicating with the computers.

B. WHY VISUAL PROGRAMMING

The challenge at hand is to bring computer capabilities, simply and usefully, to people without special training in programming. Visual programming represents a conceptually revolutionary approach to meet this challenge. This section pursues the basis for implementing Turtle Graphics in a visual programming style.

1. Dual Brain Theory

The human brain is divided into two hemispheres. For the control of movement and analysis of sensation, the assignment of duties to the two hemispheres follows a simple pattern: Each side of the brain is responsible mainly for the other side of the body. However, the distribution of the more specialized functions is quite different. Linguistic ability is dependent primarily on the left hemisphere, while the perception of melodies and nonverbal visual patterns is largely a function of the right hemisphere.

Additionally, it is generally believed that the left side of the brain thinks analytically and logically, while the right side thinks in a more intuitive and artistic sense. The left side is thought of as a sequential information processor, highly developed for verbal expressions. The right side, on the other hand, seems to be capable of more parallel processing. An image is captured as a whole. For example, when a face is seen, an immediate recognition takes place [Shu88].

Programming has always been thought of as an activity which requires the ability to think analytically, logically, and verbally. Visual programming represents a recent attempt to exploit the nonverbal capabilities of the right side of the brain.

2. A Need For a New Programming Style

Recently, the decreasing cost of computing, coupled with the widespread use of personal computers, has acted like a catalyst for more applications. By necessity, end-user computing is becoming a major trend, and expected to grow in the future. It will be extremely difficult to achieve this phenomenal rate of growth unless the style of computing evolves to such a state that a large portion of the user population can use a computer without thinking deliberately about it, much like driving any car. Thanks to the engineers who made it possible, it is not a concern with how an automobile works. Instead, energies can be spent deciding how to get from one place to another.

Learning to program in the traditional text-based languages, unfortunately, is a time-consuming and often frustrating endeavor. Moreover, even after the skill is learned, writing and testing a program is still a time consuming and labor-intensive chore. Programming has the tendency to lead to what has been termed "analysis paralysis." This refers to forgetting what the intent of the process is to produce by getting wrapped up in process of getting it out [Shu88]. It is precisely for these reasons that a Visual Dataflow Turtle Graphics Language was implemented in this research. It will provide the end-user with an intuitive, easy to learn, tool thus, allowing the user to spend more time on the critical, problem-solving thought process, rather than on the constructs and syntax of the language.

C. WHY DATAFLOW PROGRAMMING

For many years, graphs and diagrams of various sorts have been used as visual aids for the illustration or documentation of one or more aspects of the programs. But these graphical aids, for the most part, did not comprise the programs themselves. They were not executable. Until recently, the high cost of the graphical

terminals, and the large data storage needed for graphical representations, have kept the graphing and diagramming techniques on paper only.

However, the result of advances in technology and economics have made possible the incorporation of charts, graphs, and diagrams as graphical extensions of executable code.

1. Executable versus Non-Executable Diagrams

By taking a look at the traditional process of programming, multiple advantages can be seen by making charts, graphs, and diagrams executable. Traditionally, programming involves several distinct phases: problem analysis, chart or diagrammatic program depiction, translation (compiling/interpreting), and testing. And, more often than not, these processes would require several iterations at various points.

One serious problem with non-executable, visual programming aids, has to do with the need to keep both the charts or diagrams, and the code (which are basically two representations of the same program) up-to-date. It is not surprising that somewhere in the debugging process, the visual aids, no longer represents the actual code that is executed, and consequently creates problems in later maintenance of the program. Making charts or diagrams executable is an attempt to collapse the two separate processes (charting and coding) into one. This not only makes programs easier to comprehend, but also easier to document and to maintain.

Through the emulation of Prographs' dataflow paradigm, Dataflow Turtle Graphics provides a very-high-level dataflow programming tool that is directly translatable into executable code.

2. Dataflow Functionality

Dataflow languages sequence program actions by a simple data availability firing rule. When an operation's arguments are available, it is said to be "firable."

After firing, the operation's result is passed, via the diagram, to other operations which need these results as there arguments. Dataflow programs are easily integrated with larger programs through a simple diagram connection line. The diagrams present an intuitive view of the potential concurrency in the execution of the program, as well as, providing a formal meaning to the program itself.

D. ICONIC LANGUAGES

Iconic systems are made up of both visual and audible icons. While some literary systems are capable of expressing an infinite range of feelings, ideas, concepts, and thoughts, programming languages do not need the same range of expressiveness. However, through the study of literary systems, several lessons can be observed.

1. Iconic Guidelines

Since the clarity or meaning of a pictorial icon is not always apparent, it is essential to spend time with the design of icons. As with any tool, there are good pictures and bad pictures. The result of implementing the latter, may be to produce objects or concepts that are confusing or hard to remember.

Another common argument for employing graphics and icons is, that by doing so, the brain can be tapped for it's powerful pattern recognition capabilities. It must be noted, however, that the human brain is susceptible to information overload. It is important that the graphics are not so overwhelming that they can no longer be processed effectively.

Lastly, providing access to an icon's definition is extremely important and necessary for an iconic language. A dictionary must be provided for the potential vast number of pictures in an iconic system. When the number of icons is small, it is possible to have them presented on the screen so that the user can point to the one desired. This method is not possible when the number of icons is large.

Iconic programming languages provide an incentive to learning. Pictures provide the user with challenge, fantasy, and curiosity. It is for these reasons that all visual systems must incorporate some level of iconic programming.

E. TURTLE GRAPHICS DESIGN AND IMPLEMENTATION

The design and implementation of Dataflow Turtle Graphics is centered upon creating a graphics programming tool that combines the benefits of visual dataflow programming, including the use of icons, with the extremely successful concept of Turtle Graphics. By exploiting the benefits of the chosen programming style, it is possible to increase the benefits gained through Turtle Graphics.

The system is designed for the user to interface through a windowing environment utilizing a mouse. Additionally, it provides standard Macintosh editing functionality, and on-line Help. All Menu options are supported by "Hot Keys."

The design and implementation for this research was basically two-fold. First, it was necessary to create a Turtle object, with specific attributes, that could be defined by the user. Having created this Turtle object, it was then necessary to define how this object should respond, behaviorally, to specifics messages. These methods of behavior include, but are not limited to: forward, turnto, turnright, ect., and are fully defined in Appendix A.

At the completion of this implementation phase, the user was required to interact directly with the Prograph programming environment to program the Turtle. This led to the second, and most demanding, phase of the implementation process.

To remove the necessity of the user being required to learn Prograph, it was necessary to integrate the Turtle Graphics code with a portion of code from Armedeus[1]. The successful completion of this code integration exemplified the

1. Armedeus is a visual, object-oriented database, thesis developed by several students under advisement by C. Thomas Wu, Prof., Computer Science Department, Naval Postgraduate School, Monterey, Ca.

object-oriented concepts of developing reusable, sharable, integratable, and extendable code.

1. Developing Turtle Class/Objects and Methods

The design of the Turtle Class was driven, primarily, by what behaviors the objects of the class were required to perform. General behaviors included the ability to draw graphics, through various manipulations, on the screen. The code for this phase of implementation was developed and tested prior to integration.

a. Class Hierarchy

The design of the Class hierarchy was based on the need for creating Turtle objects, as well as, the requirement to develope a user interface to define and test the graphics tool. Figure 3.1 shows the graphical class hierarchy for the first phase of implementation.

The Turtle Class would be the template class for all Turtle objects created thereafter. It would include the necessary attributes and methods common to all Turtle objects. The pTurtle Class contains an augmented list of methods and attributes, necessary to respond to more complex user commands.

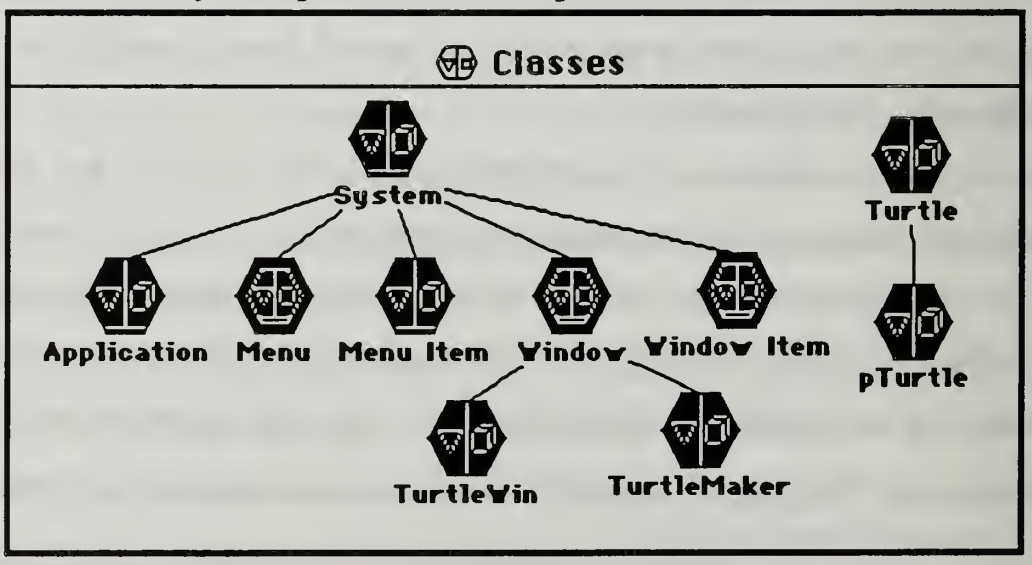


Figure 3.1 Class Hierarchy

The user interface was developed to define the Turtle object, and for displaying all graphics. TurtleWin and TurtleMaker Classes provide the templates for the user interface needs. The general function of these interface windows, for the first phase of implementation, is provided below. Final interface-window functionality will be provided with the discussion of the implementation of the second phase.

b. Turtle/pTurtle Class Definitions

Figure 3.2 depicts the graphical representation of Turtle Class. It contains the necessary framework to define specific Turtle instances. Additionally, in this implementation, it serves as the superclass for the subclass pTurtle.

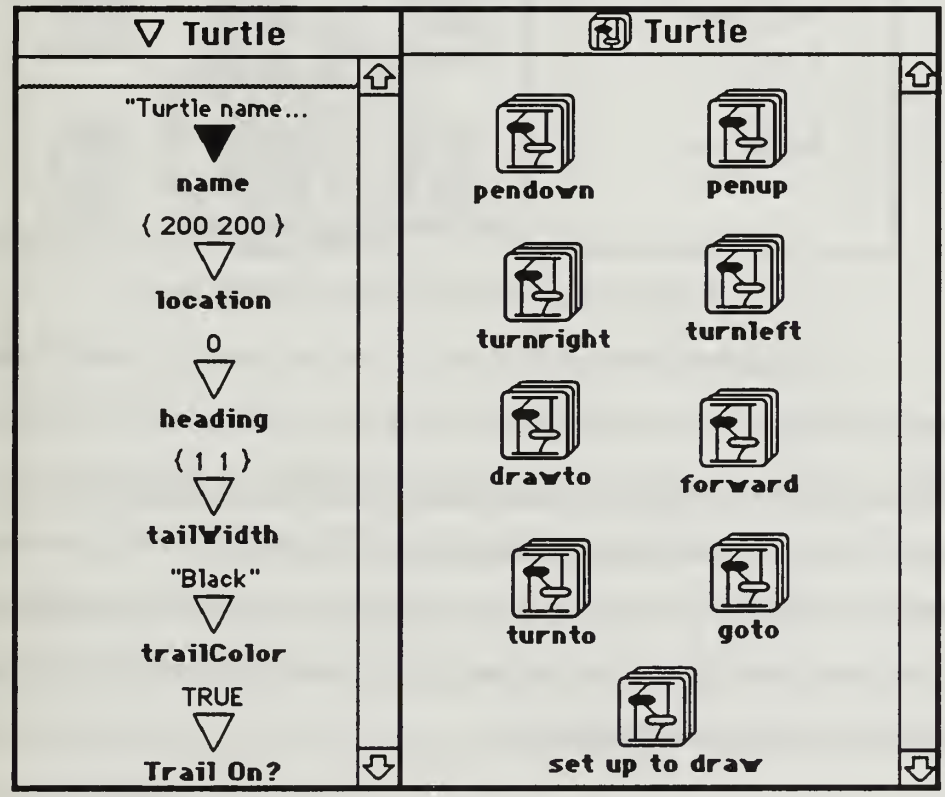


Figure 3.2 Turtle Class Definition

Figure 3.3 shows the graphical representation for pTurtle class. Although all attributes are not shown, pTurtle class has inherited all Turtle class

attributes. Additionally, it has augmented its attribute list with the attribute *program,* and its methods with several complex methods to expand the users list of available commands.

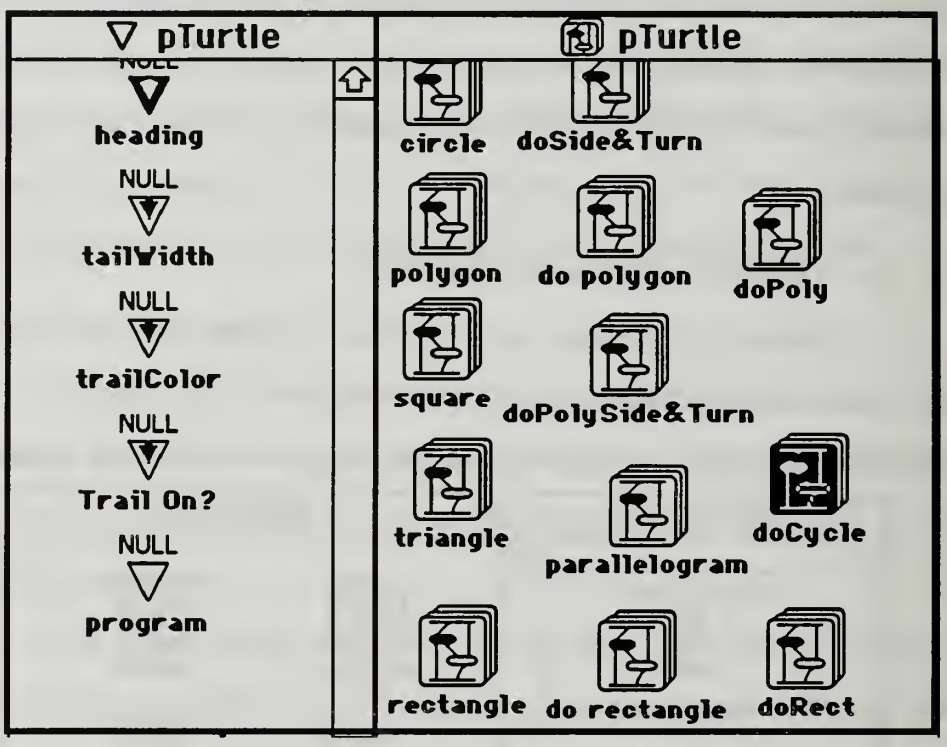


Figure 3.3 pTurtle Class Definition

To support drawing routines, it was necessary for each Turtle instance to know some basic information about itself. At a minimum, the Turtle object had to know its location on the drawing screen, the heading or direction it was going, and its name. To give the user some additional control over the Turtle, pen-characteristic controls were provided. These controls include, setting the pen-width, setting the pen-color, and turning the pen on and off. A more complete definition of these methods is provided in Appendix A.

c. *User Interface Design and Implementation: First Phase*

Developing an intuitive user interface should be the goal of any interactive program. The interface design for Dataflow Turtle Graphics requires the

user to interact in a windowing environment using a mouse. The system provides standard Macintosh editing functionality, as well as, on-line help and "Hot Keys" for all Menu options.

The first phase of implementation consisted of three general interface windows and a menu bar: Turtle Maker, Turtle Display, Prograph's method-programming windows, and File, Edit, and Turtles menu options.

Figure 3.4 shows the Turtle Display window which displays the graphics routines that have been programmed. The Draw button serves two purposes

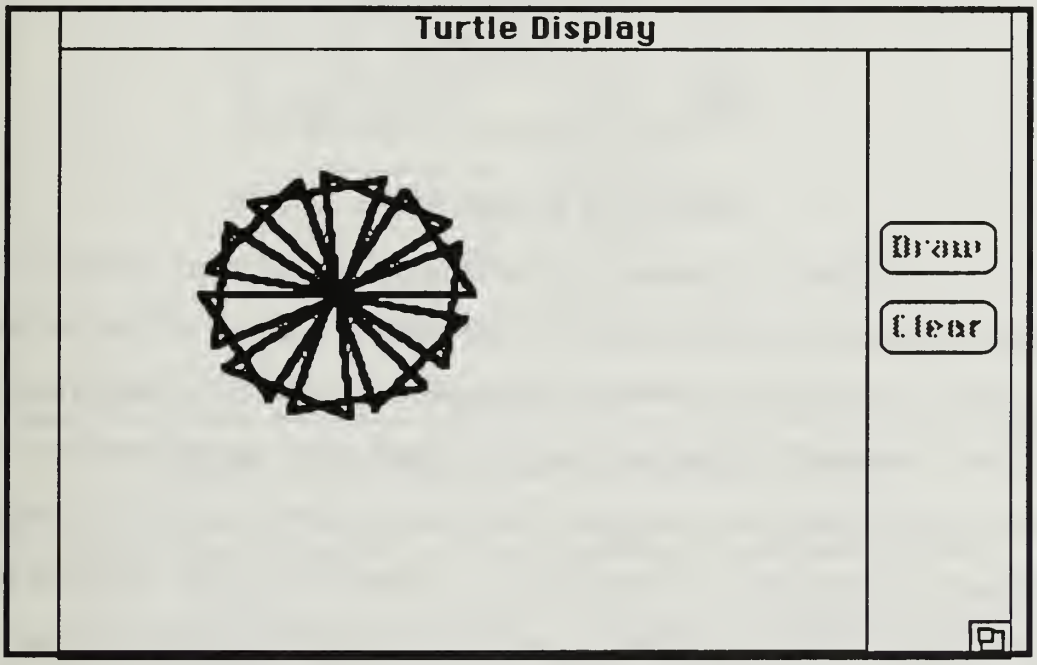


Figure 3.4 Graphics Display Window

by initiating the drawing of all programs that have their "Trail On?" switches set, (see Figure 3.6), and also by activating Prographs' method programming window for those Turtles that have not yet been programmed. The Clear button simply clears the graphics screen.

Figure 3.5 shows the Turtles menu options. The *New* option will open the Turtle Maker interface window (see Figure 3.6), allowing the user to set specific Turtle attributes. The *Delete* option removes one of the listed Turtle programs that

appear below the gray line. The check-symbol, appearing next to the name implies that the Turtles' "Trail On?" attribute is set, and represents whether a program is active for drawing purposes.

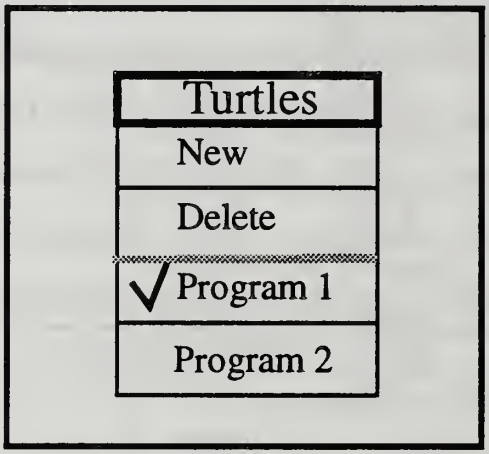


Figure 3.5 Turtles Menu Option

Figure 3.6 shows the Turtle-definition interface window. This is opened when the *New* option in the Turtles menu is selected, or if any of the listed programs is selected. This provides an editable environment for setting or changing the name, heading, location, pen-size (Trail Width), pen-on/off (Trail On?), and pen style or color (Trail Color) attributes. The result of depressing the OK button sets the attributes of a previously defined Turtle, or creates and sets the attributes of a new Turtle object. Cancel button simply cancels the operation and returns to the graphics display window. The format for entering location information is "{vertical-offset horizontal-offset}". The origin of the screen, {0 0}, is located at the upper-left - hand, corner of the display screen. Turtle heading information is entered as a positive integer value, 0 - 360. Compass heading relations are: North-0, South-180, East-90, and West-270.

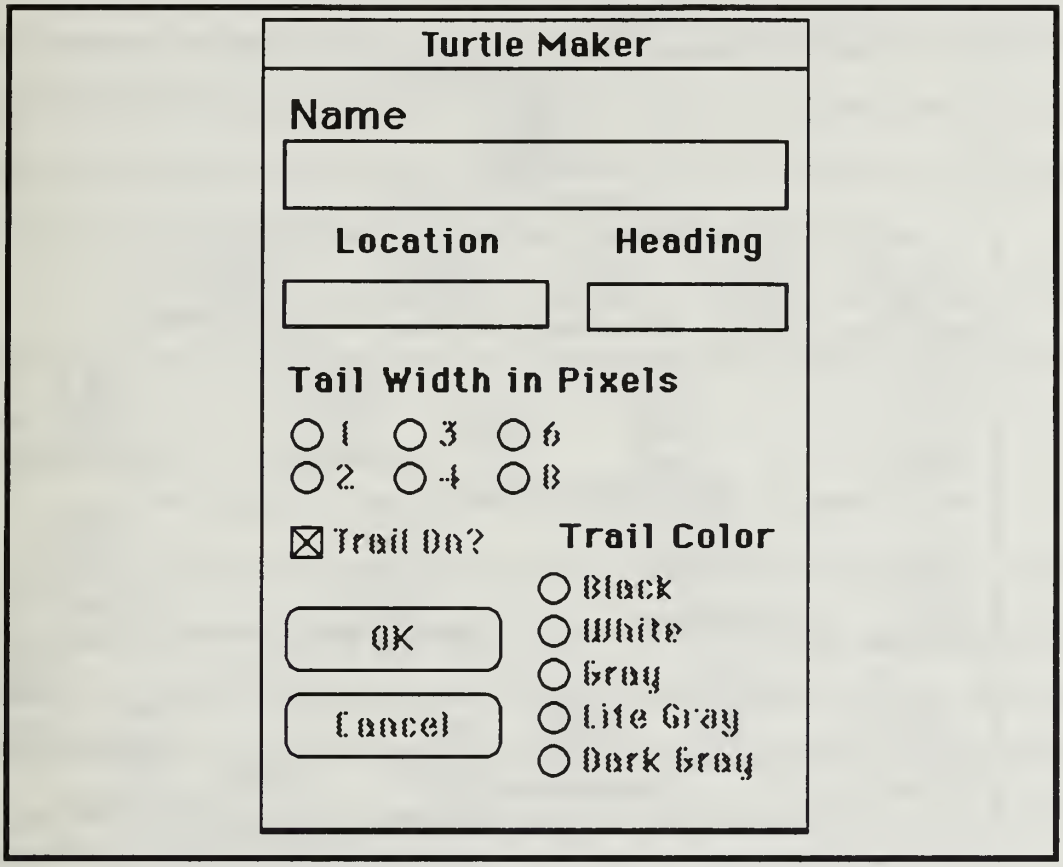


Figure 3.6 Turtle Attribute Interface Window

2. Integration of Turtle Code and Dataflow Programming Code

After completing the first phase of design and implementation, it was necessary to relieve the user from the requirement of Turtle programming within Prographs' method-definition window. Armedeus provided the necessary visual dataflow programming environment to be integrated with the completed Turtle Graphic code.

a. Class Hierarchy

Figure 3.7 shows the revised class structure for the final Dataflow Turtle Graphics language. The obvious goal was to maximize the benefits of the object-oriented design by taking a "black box" approach with the code integration.

Turtle hierarchy remained disconnected from the Dataflow hierarchy, and new subclasses were created as required.

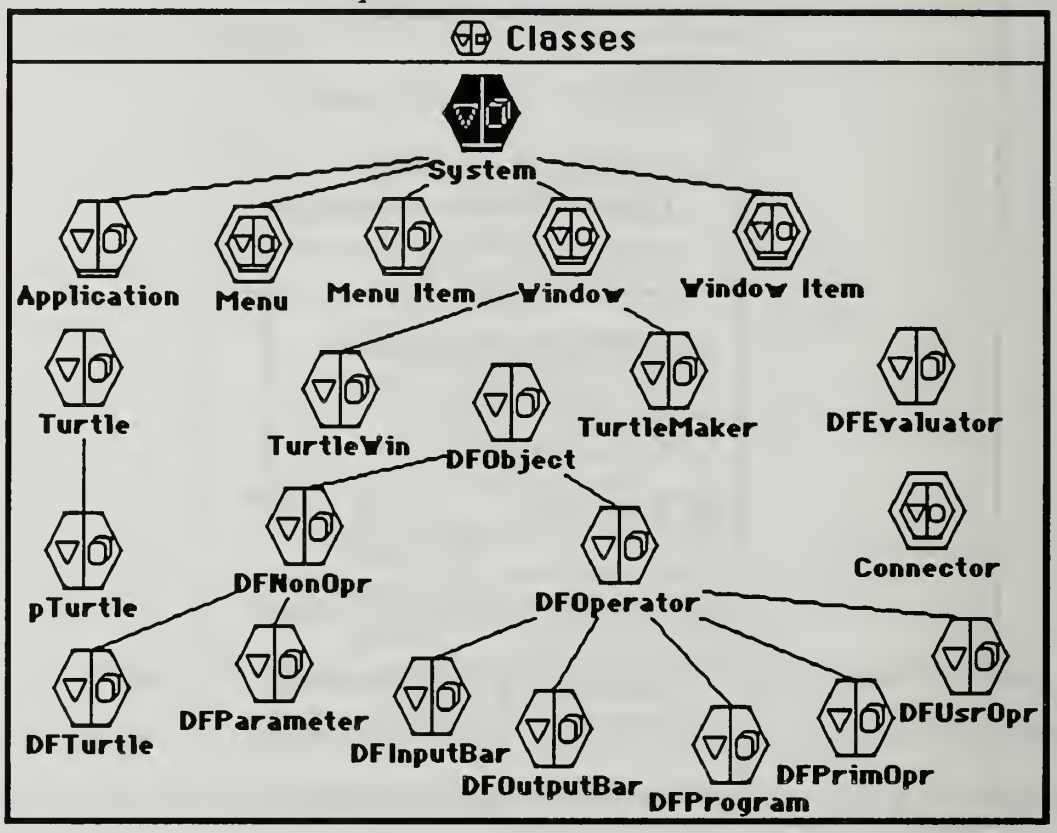


Figure 3.7 Dataflow Turtle Graphics Class Hierarchy

The Dataflow system, in general, provided the windowing environment for the programming interface, and the code for the interpretation of the programmed objects. Some changes in the existing code was necessary to support the correct translation of new programming objects. However, the intent of this research was not to re-design the code provided by Armedeus, but rather to re-use as much of the existing code as possible to efficiently create a functional Dataflow Turtle Graphics prototype, thus maximizing the benefits of an object-oriented design.

b. DFObject and Descendents

DFObject represents the superclass for all programming objects. It has two immediate descendents, DFOperator and DFNonOpr. DFOperator represents

the superclass for all objects that translate into either predefined primitive operators/ user commands or user defined operators/commands. Additional descendents include input bar, output bar, and DFProgram. These latter three objects remain in the design and implementation phase and are not completely functional. All of these objects have single or multiple input values and a single output value.

DFNonOpr represents those objects that do not translate into graphics commands. These objects have only a single output value which is an integer or a Turtle object.

c. *User Interface Design and Implementation: Second Phase*

The second phase of design and interface implementation consists of three general interface windows; Turtle Maker, Turtle Display, and Manipulation Window. The Menu bar consists of File, Edit, and Turtles options. The major interface changes required the incorporation of Armedeus' Manipulation Window for dataflow programming, and removing some of the functionality of the old Turtle Display window. The Turtle Maker interface window remained unchanged, however the means of accessing it has expanded to include double-clicking a Turtle object on the Manipulation Window.

Figure 3.8 is the Manipulation Window used for programming the Turtle objects. It contains several function buttons, and a programming pad. Generic input programming objects are place on the pad by clicking anywhere on that pad. After typing the desired object name (turtle name, command name, or integer value) into the generic input object, an appropriate object icon will replace the generic object. The object "Sam" shows the icon representing a Turtle object. Objects may be removed from the programming pad by clicking on them, to highlight them, and depressing the keyboard Delete button. Manipulation Window button functionality follows:

Display Program will provide a listing of previously defined programs for the user select to be displayed on the programming pad. ***Save Program*** prompts the user to name the displayed program, and saves that program for further reference. ***Delete Program*** provides a listing of saved programs, and prompts the user to select one for deletion. ***Clear*** simply clears the programming pad. ***Redraw*** will refresh the programming pad with the latest displayed program. ***Define Primitive*** is not completely functional, but it provides the user the ability to define a new primitive command. The user will be prompted to name the new command, after which the programming pad will be augmented with the appropriate number of input bars and an output bar. The user will then finish the dataflow program and depress the ***Save Primitive*** button. This will save the displayed program as a new user command for further use. Additionally, the Help listing will be updated to include this latest command. The Draw-Turtle-Icon button represents the function of drawing or executing the displayed program.

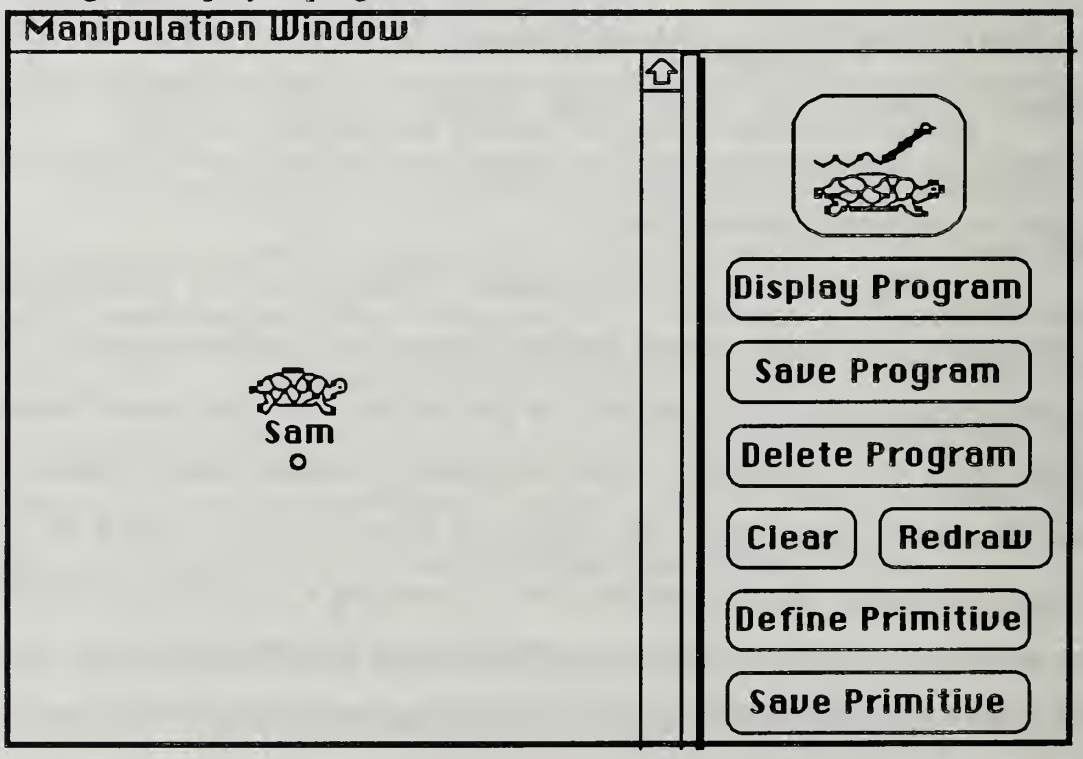


Figure 3.8 Dataflow Programming Window

Figure 3.9 is the revised Turtle Display window. It is activated by depressing the *Draw-Turtle-Icon* button. Much of the original functionality of this window was moved to the Manipulation Window, since Turtle Display window was no longer the main program-window. The *OK* button clears and returns the operation back to the Manipulation Window for new or revised programming. The *Print* button will print the displayed graphics.

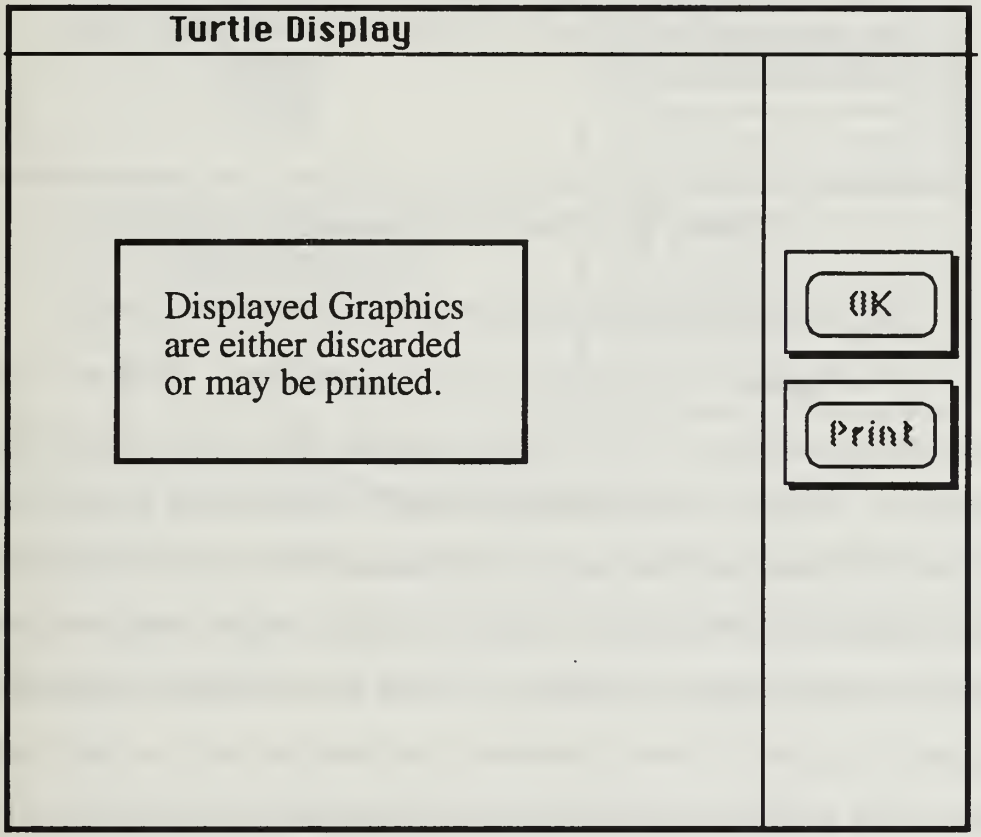


Figure 3.9 Revised Turtle Display Window

One last interface window, not previously discussed, is the Help window. This window, although accessible to the user, does not yet contain command help information, nor is it automatically augmented with the creation of

new user defined commands. Figure 3.10 shows this interface and explains its design and implementation.

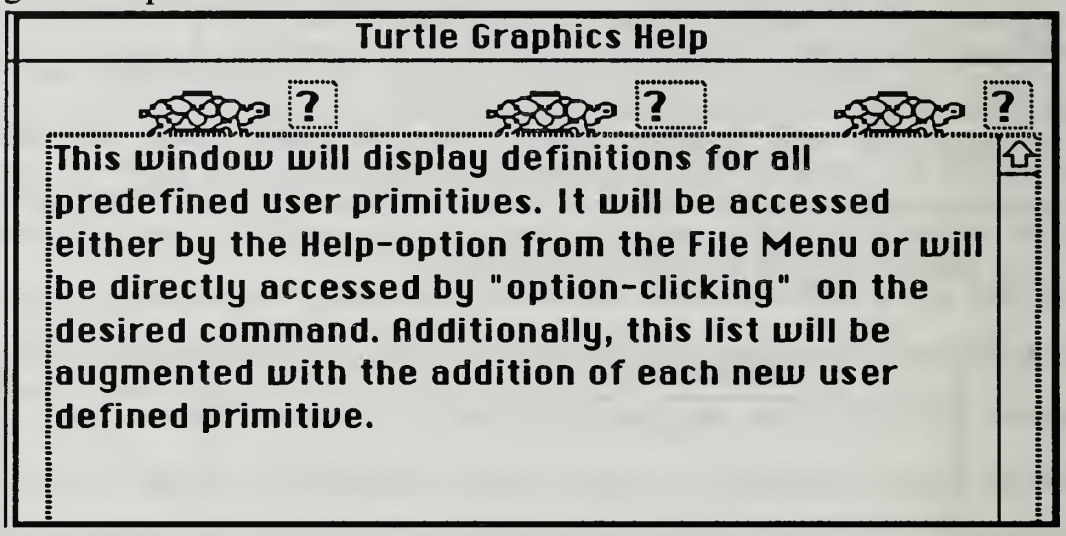


Figure 3.10 Turtle Graphics Help Window

d. Program Objects: Icon-Description and Functionality

Figure 3.11 shows the Manipulation Window with various programming objects. The lettering of each object was added for clarity of discussion. Object A, as stated previously, represents a programmable Turtle Object. The Turtle icon has only a single output value, since it does not require any input information for execution. Double clicking on this object will automatically open the Turtle's Maker-definition window for reviewing or editing. Object B, represents an integer value parameter, and also has only a single output value. Object C, is a typical predefined operator/command, and is represented by a "black box" with the appropriate number of input terminals and an output terminal. Double-clicking on any predefined operator will open the Help window to review its definition. Object D represents an operation that encapsulates a program. It's icon consists of multiple black boxes, and has no inputs or output, since it represents a stored programmed. Figure 3.12 shows the result of double-clicking on any

program-operator. This is the actual code that defines the program and can be edited as required. Object E represents a user defined operator. Its icon is identical to any

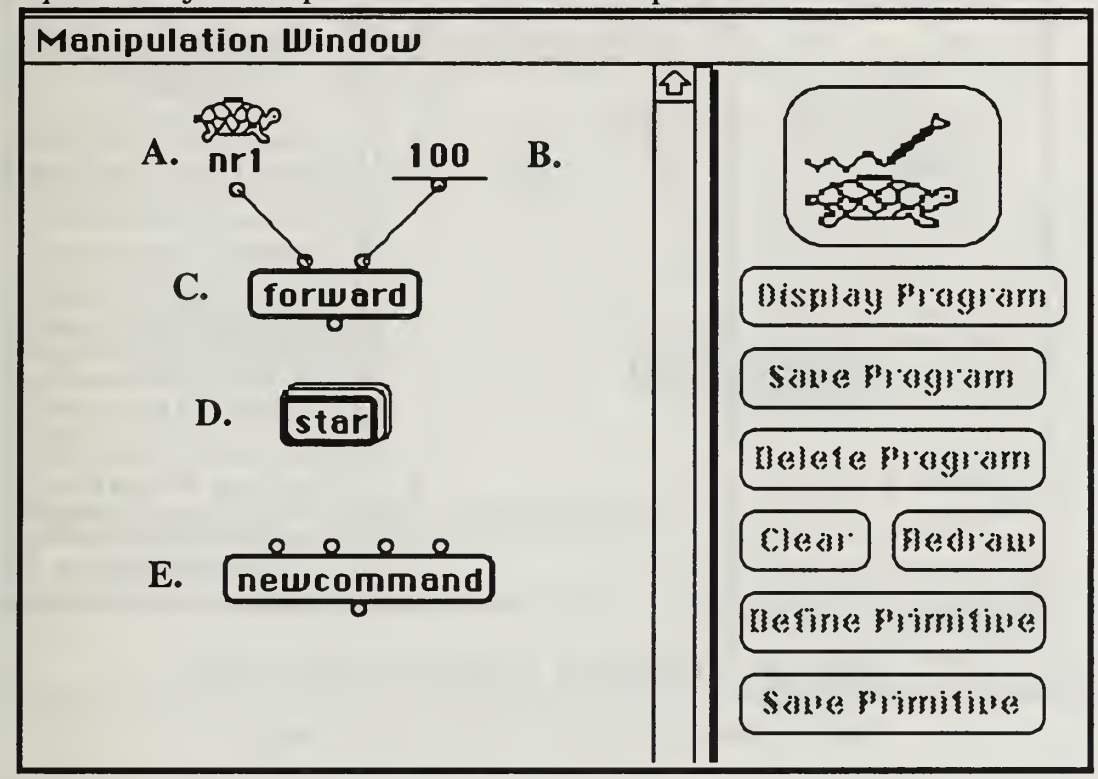


Figure 3.11 Programming Object Icons

predefined operator, since it functions in the same manner as a predefined operator. However, when these objects are double-clicked, their associated program window opens for editing or reviewing. Figure 3.13 shows the code for the *newcommand* operator.

Turtles are programmed by connecting dataflow lines between appropriate roots and terminals. This is achieved by clicking on either of these objects, then clicking to the point of connection. Programmers are prevented from connecting root-to-root or terminal-to-terminal, and are provided a warning message.

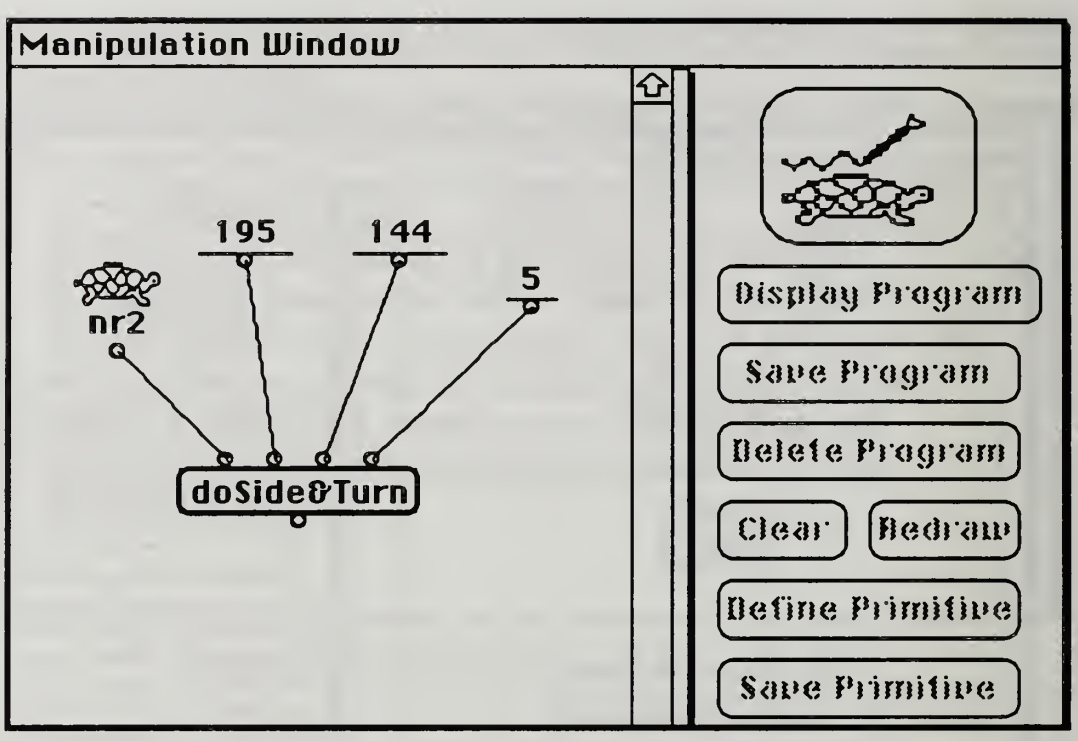


Figure 3.12 Stored Program Code for *Star*

Appendix A provides detailed definitions for all predefined user commands. Chapter IV will show how to use these tools to produce a Dataflow Turtle Graphics solution for a particular project.

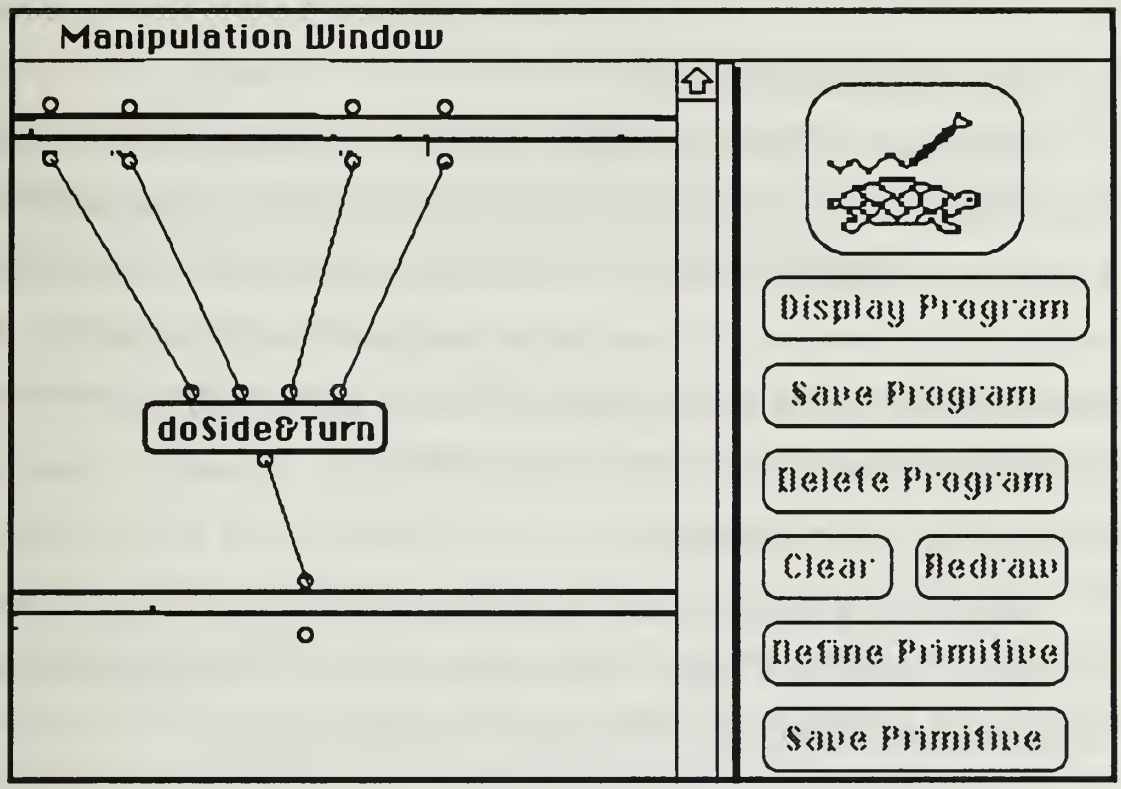


Figure 3.13 User-Defined Operator Code

IV. PROBLEM SOLVING WITH DATAFLOW TURTLE GRAPHICS

A. GENERAL DISCUSSION

The primary purpose of this chapter is to show how to utilize the tools provided in this thesis to solve a particular problem. It is assumed that the user has the basic knowledge of Prograph to activate the Turtle Graphics Application. Additionally, the user will be required to save any programming done in Turtle Graphics, at the Prograph prompt, when quitting. Lastly, all figures show the actual programming and output windows from the functional prototype.

B. PROBLEM STATEMENT

The problem at hand is to create a very basic Dataflow Turtle Graphics solution to display a picture of a "man". Keep in mind that the crude graphics are not as important as are the steps that were taken to complete the project.

C. DEVELOPING A SOLUTION

First, and foremost, there is no single correct solution to this problem. The approach to the solution herein attempts to follow the same concepts that have been brought forward in this research. The limits of the functionality of the prototype, in some cases, has required relatively complex coding to achieve a simple result.

1. DFTG's Object-Oriented Approach to Problem Solving

Dataflow Turtle Graphics provides an intuitive tool, in the turtle metaphor, for programming specific components of an overall solution. The turtle class represents a logical collection of abstract turtle objects instead of subprograms, as in earlier developments of Turtle Graphic languages. The flexible and intuitive nature of Dataflow Turtle Graphics provides for a sound object-oriented programming solution to problems. DFTG allows the user to modularize, or partition, the problem into individual components, thus reducing the overall complexity by creating a

number of well-defined boundaries within the program. The development life cycle, using DFTG, emphasizes the incremental, iterative development of a solution. The intent is to design, program, and test components separately. As components are completed, they are integrated until the entire solution has been programmed. With this approach, there is never a big-bang event of system integration.

2. "Man-Project" Problem Reduction

The generic image of a "man" object will be divided into the following separate components: Head, Face, Body, Arms, Legs, and a Bowtie. Each component will be programmed with it's own individual Turtle object, with the exception of the arms and legs, which will use the same Turtle. Separate Turtles allow for individual characteristics and controls over that specific part. Additionally, separate turtle objects offer the abstraction benefit for developing these individual components. As components are completed they will be integrated together to solve the overall project.

a. Create Turtles

The first thing to do is to create some new Turtle objects. This is accomplished by selecting *New* from the Turtles Menu. Each Turtle will be uniquely named and defined.

b. Creating the Head

Figure 4.1 represents the code to display a crude *head* object. To execute the displayed program, depress the Turtle Draw Icon. When satisfied with the displayed code simply depress the Save Program button. Name this program, *head*, at the prompt. This, in affect, has encapsulated the displayed code in a single command.

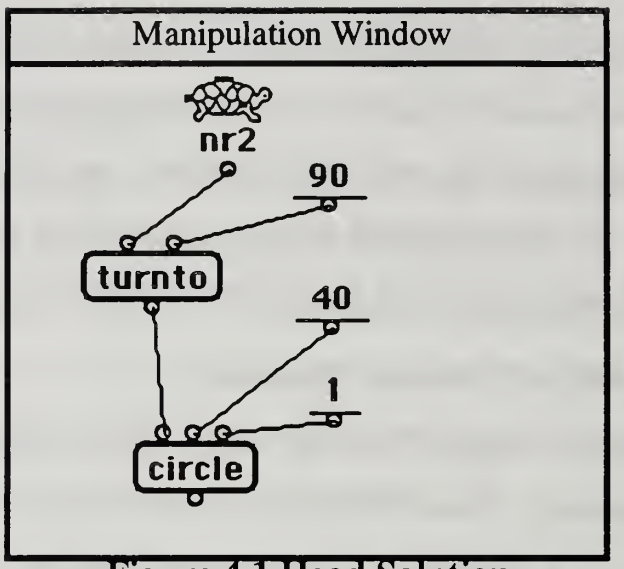


Figure 4.1 Head Solution

c. *Create a Face*

The next step is to create a *face* for the *head* object. Figure 4.2 shows the code to create this face. This Turtle is responsible for drawing the eyes and mouth. This code shows clearly, the need for a "move" operation that has the same result as a "forward" operation with no drawing. Again, when satisfied, save this code as *face*.

Figure 4.3 shows the encapsulated code, and the results of executing the *head* and *face* routines. Any changes, made to the individual routines to satisfy integration, must be saved at the prompt.

d. *Create a body*

Figure 4.4 shows the code to create a *body* for this project. It consists of a neck and trunk shape. Again, when the solution for the body is acceptable, save this program as *body*.

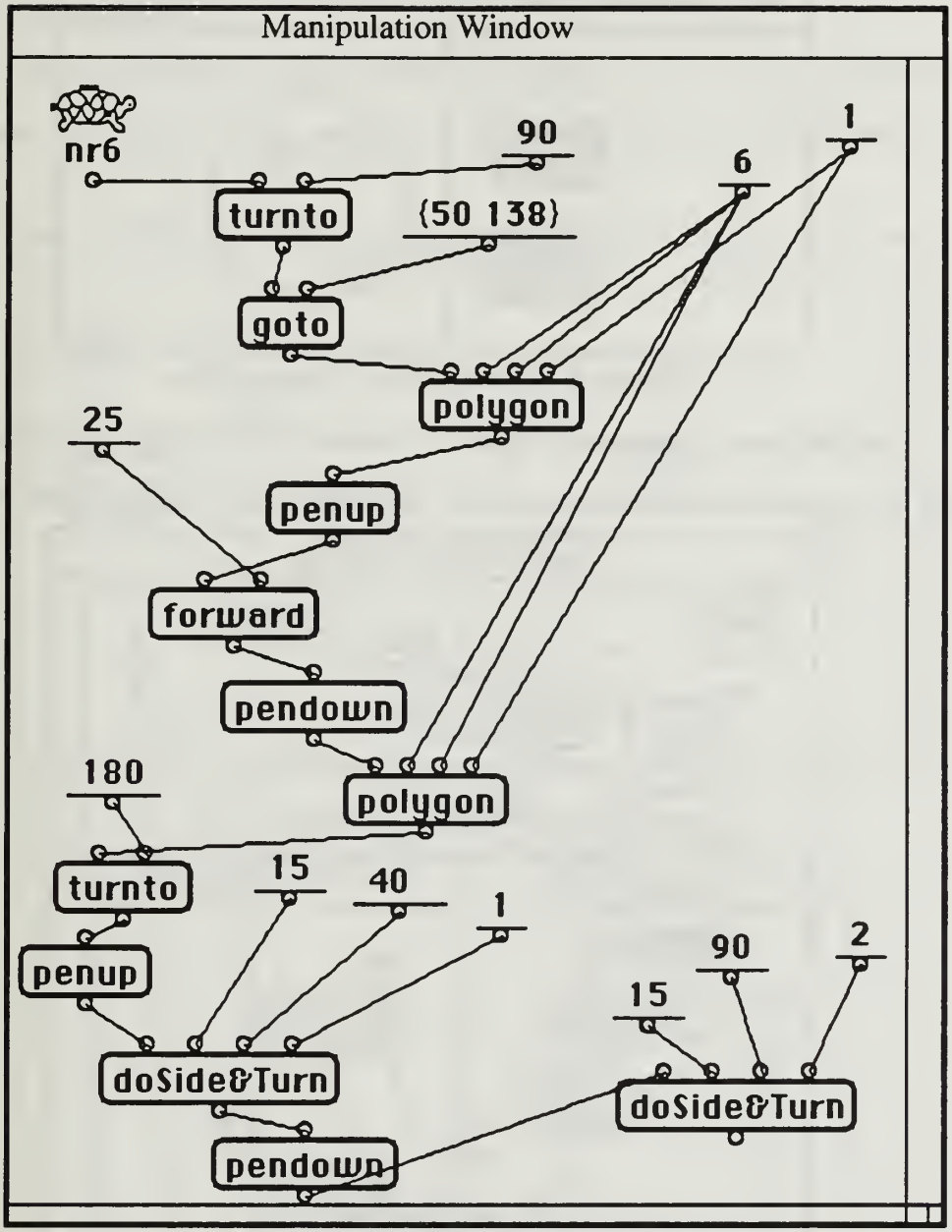


Figure 4.2Face Solution

e. Create a bowtie

Figure 4.5 shows the code for the *bowtie*. Save this program as *bowtie*. Figure 4.6 shows the integrated code for the bowtie and body parts, as well as, the result of this execution.

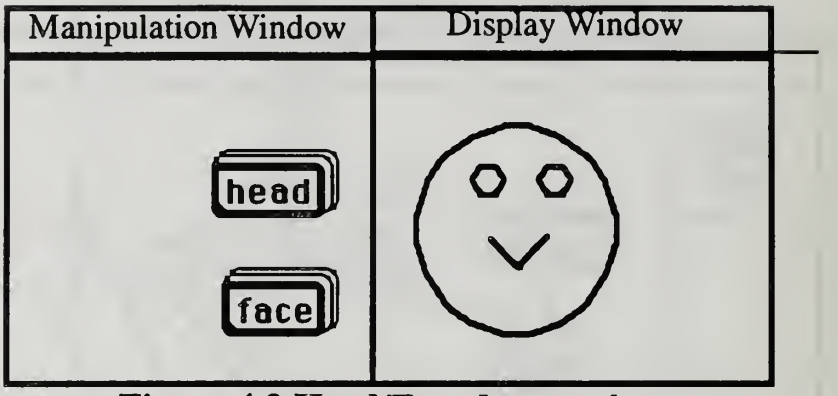


Figure 4.3 Head/Face Integration

At this point, there exists solution routines for head, face, body, and bowtie. Figure 4.7 shows the code and results of executing these routines together.

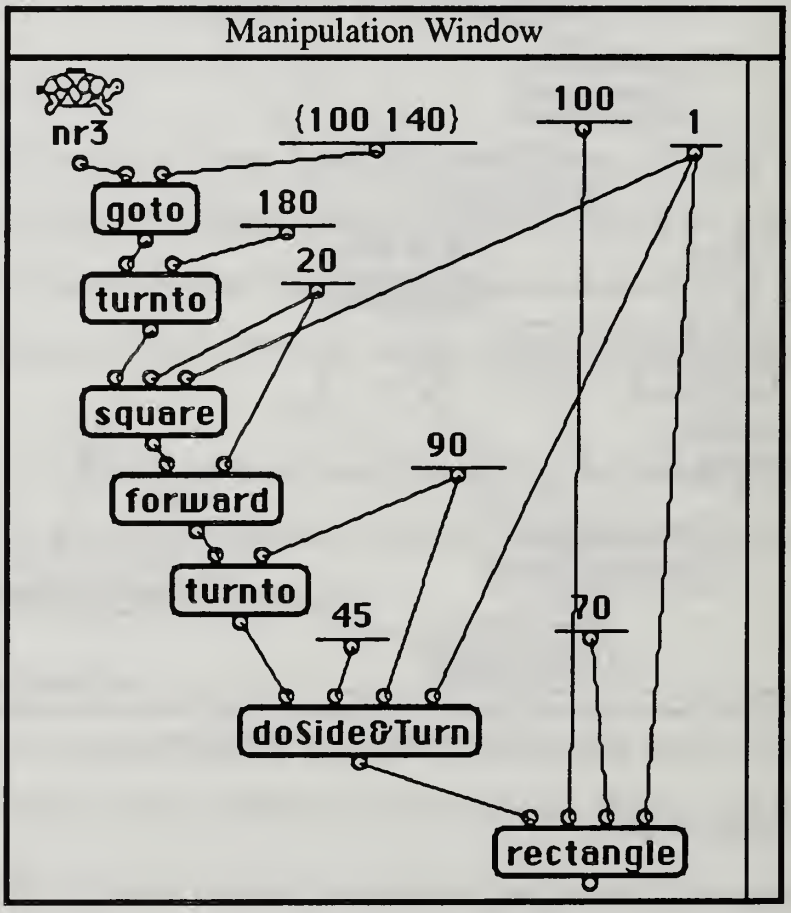


Figure 4.4 Body Solution

f. Create legs

Figure 4.8 shows the code for the creating *legs* for the man object. Figure 4.9 shows the results of integrating the previously defined routines with the *legs* routine.

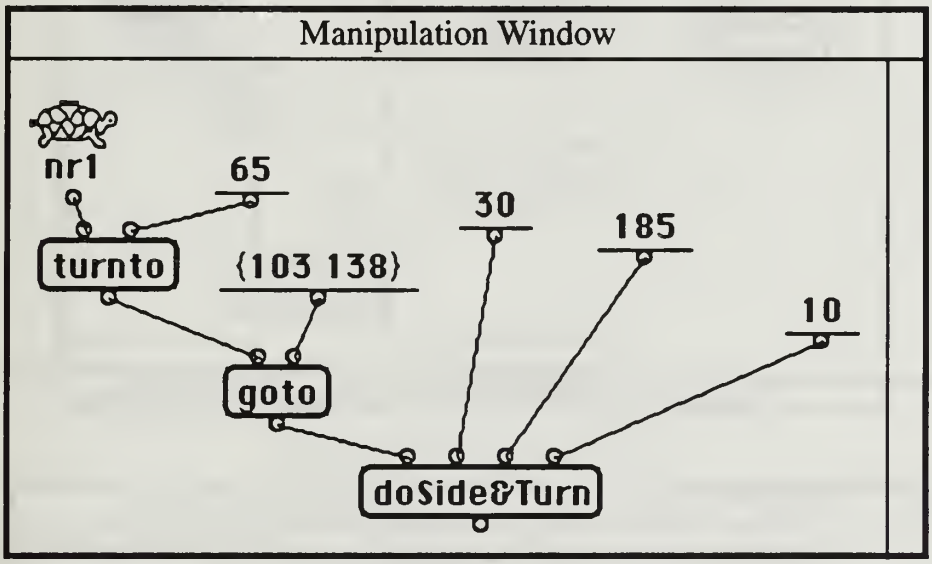


Figure 4.5Bowtie Solution

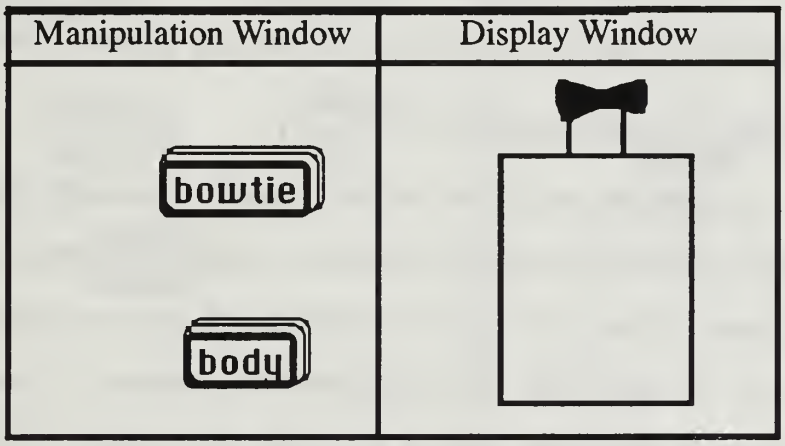


Figure 4.6Integrated Body/Bowtie Solution

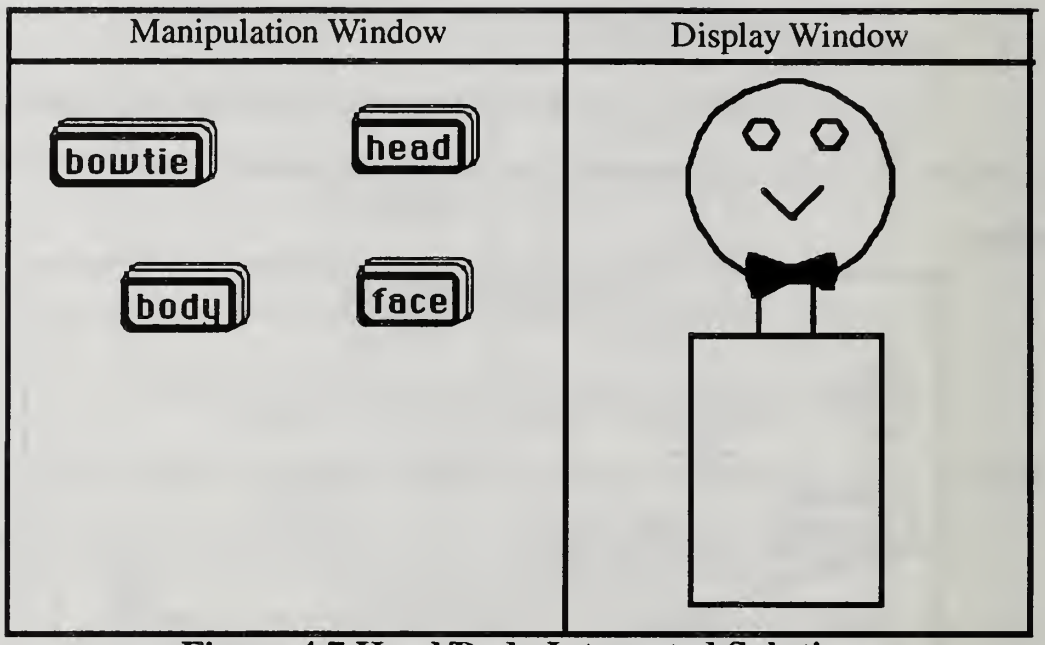


Figure 4.7 Head/Body Integrated Solution

g. Create arms

Creating the *arms* will complete the modular development of the "man" project. Figure 4.10 shows the code to display arms for the man. Figure 4.11 shows the integration of the previously defined routines and the arms routine.

h. Final Code Encapsulation

In order to display the project, "man", in a single command, depress Save Program while all subroutine calls are displayed. Enter *man* at the name-program prompt. Figure 4.12 shows the new call to the encapsulated man routine. This now can be used with other encapsulated routines to develope additional displays. Keep in mind, the idea is to break up a large problem into smaller, manageable components, complete or solve the smaller components, and then integrate the completed routines to solve the larger problem. It's a simple, yet powerful problem solving strategy, that can be reinforced and refined through the use of Dataflow Turtle Graphics Programming.

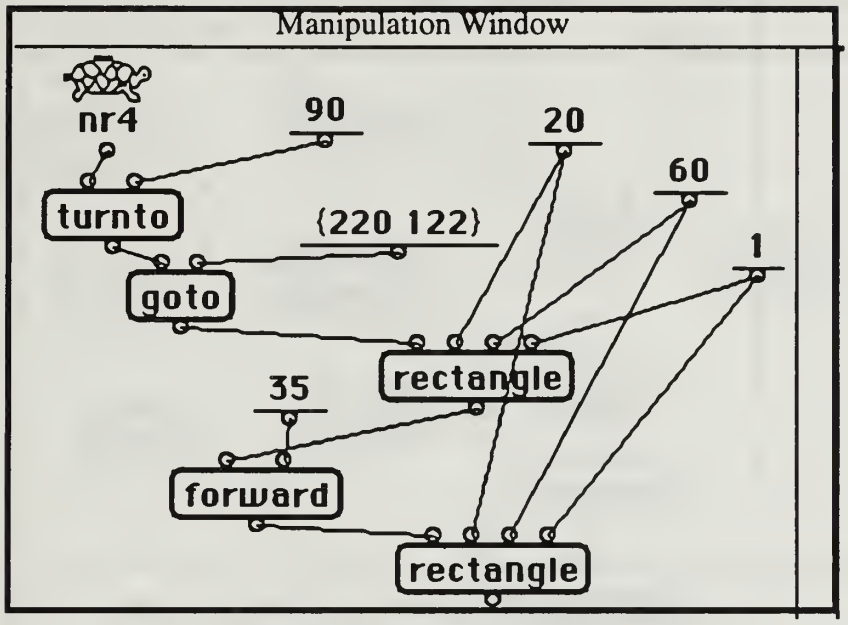


Figure 4.8 Legs Solution

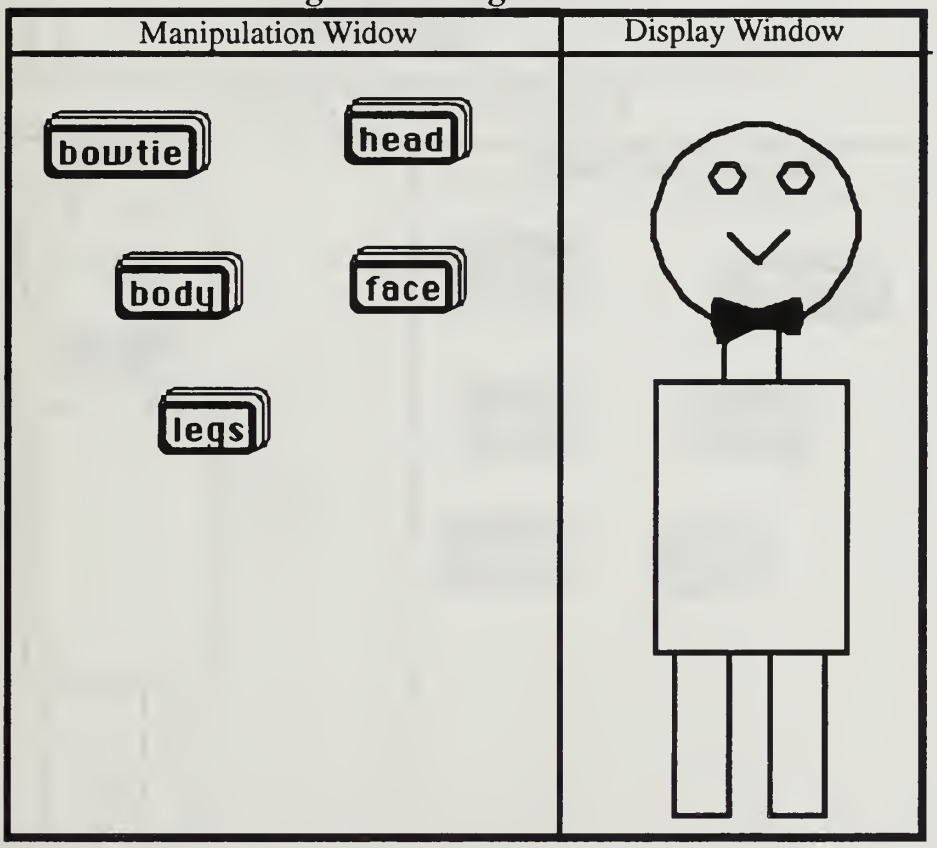


Figure 4.9 Head/Body/Legs Integration

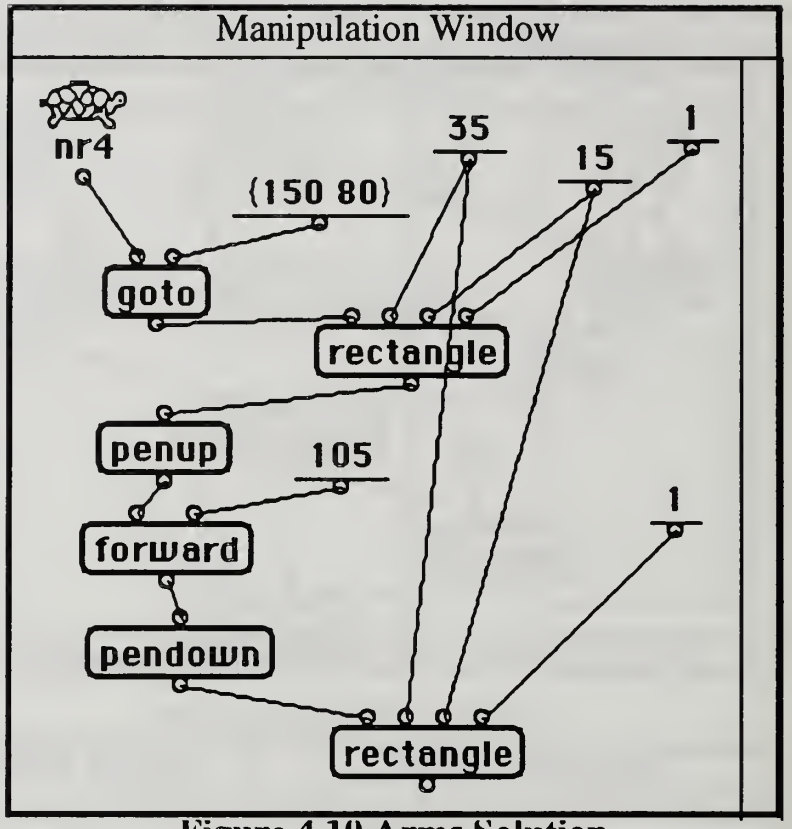


Figure 4.10 Arms Solution

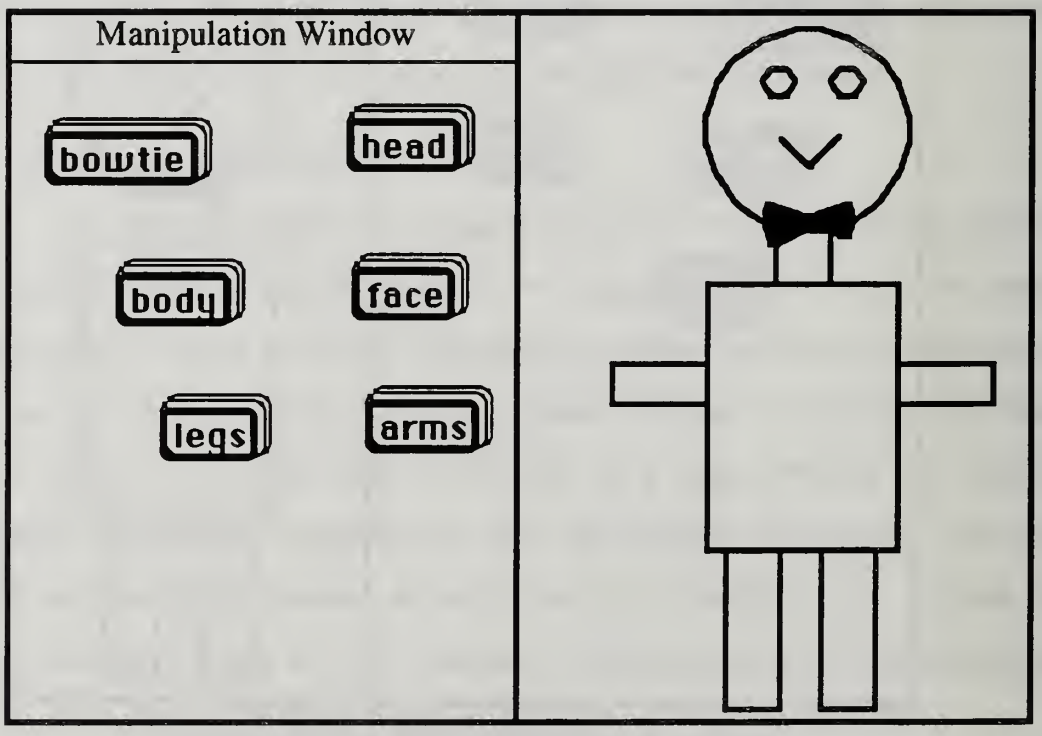


Figure 4.11Complete Integration

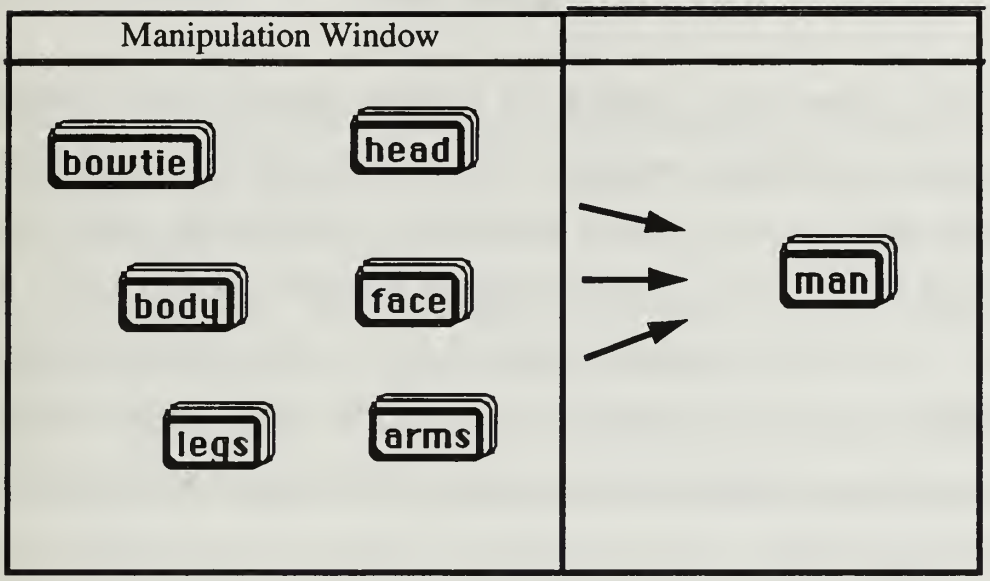


Figure 4.12 Final Man Encapsulation

V. SUMMARY, CONCLUSIONS, & SUGGESTIONS FOR FUTURE RESEARCH

The purpose of this research was to design and implement a Dataflow Turtle Graphics programming language for children to use to develop their problem solving skills, as well as their fundamental programming skills. This research provides the first stage of development for a complete Turtle Graphics Language. There was no related research locally, prior to this implementation of Turtle Graphics, however this project does use some special purpose code, dataflow programming environment, developed in another research project, Armedeus[1].

A. SUMMARY

In summary, a complete literature review was accomplished in which Object-Oriented programming, Logo's Turtle Graphics programming, and the Dataflow Programming Language Prograph were researched. The design and specifications for developing a Dataflow Turtle Graphics Language was reviewed, and an object-oriented prototype was implemented.

This research development has come from the intersection of a multiplicity of ideas including: Object-Oriented program design, Turtle Graphics, and Visual Dataflow Programming. The proposed combination of concepts presented in this research provide a new and exciting tool. It is characterized by being generally accessible, and offering a service perceived as being usable and useful to a variety of users.

---

1. Armedeus is a visual, object-oriented database, thesis developed by several students under advisement by C. Thomas Wu, Prof., Computer Science Department, Naval Postgraduate School, Monterey, Ca.

B. CONCLUSIONS

The strongest indicator of Turtle Graphics effectiveness is that studies are conducted on how it helps children. In itself this demonstrates it's wide recognition. Unlike a multitude of educational software, versions of Turtle Graphics languages have evolved over the past 30 years, and is as worthwhile now as when it was first introduced. Along with the development of Logo language, comes the encouragement users require to explore, learn, and think.

Dataflow Turtle Graphics, through the combination of the Turtle-metaphor in Turtle Graphics and Dataflow programming, provides users with an intuitive tool that allows them to spend less time learning the linguistic constructs and syntax of the programming language, and more time on the key issue of problem solving. This is primarily do to visual style of programming required to solve problems. The visual implementation of this prototype language, including the dataflow paradigm and iconic constructs, is easy to learn and offers new users the power to program relatively complex graphics routines in short order. It provides users with a programming tool that is more in-line with the natural way of thinking.

Although DFTG is not a complete programming language, it provides the necessary features to argue for further development of a fully complete Dataflow Turtle Graphics Language.

C. SUGGESTIONS FOR FUTURE RESEARCH

Future research in this area should include, but is not limited to, the following areas: completion of "user-defined Turtle command" functionality, completion of "user-help" functionality, implementation of additional Turtle functionality, expansion of language control constructs, implementation of more complete error detecting capabilities, and the incorporation of a programming pallet of available commands to speed up and simplify the programming process.

1. **Completion of "user-defined Turtle command" functionality**

This programming feature is necessary for all languages. Although this feature is not completely functional, this prototype does show the added power and flexibility that it would provide. User-defined commands provide a means for programmers to fully explore their creative problem solving skills.

2. **Completion of "user-help" functionality**

DFTG has provided the necessary windowing interface for Help support, however the complete command definitions need to be incorporated. Expanded functionality would include, the ability to access any command-definition directly without browsing through the entire dictionary, and the ability to augment the dictionary with the creation of new user-defined commands.

3. **Expand language control constructs**

This prototype is limited in its control constructs. The user needs to be able to control the flow of its program throughout. At a minimum, DFTG should be able to support looping, next-case, and termination capabilities. At present, DFTG's complex predefined commands, such as square, polygon, doSide&Turn, ect., provide an iterative function through their right most terminals.

4. **Fully implement Error detection/correction capabilities**

Error messages should be clear and informative. Warning messages should provide the user with the ability to correct a situation before program execution. It is sometimes desirable to be provide a warning message prior to saving changes to an existing program or deleting portions of code. Additionally, it might be especially helpful to be warned when duplicate names are being used for different programs. The bottomline is that, DFTG needs to provide a friendly and forgiving environment

for programming. This, in and of itself, will encourage the user to continue to use DFTG.

5. Incorporate a programming pallet of available commands

Programming pallets have shown to be quite useful in graphics applications. Providing a pallet of user commands would provide a means to simplify writing programs, as well as speed up project development.

6. Implement additional Turtle functionality

DFTG's object-oriented design has left the door open for adding new Turtle functionality. This may include, but is not limited to, adding math symbols to create math functions, using sound for creating music, and incorporating pictures for creating visual story books.

7. Perform statistical studies of user effectiveness

Do to the limited functionality of the present implementation of Dataflow Turtle Graphics, it was not possible to include user studies to substantiate it's effectiveness. As the functionality of Dataflow Turtle Graphics expands, there should be an in depth analysis of the actual effectiveness of this Turtle Graphics versus the traditional text-based versions of Turtle Graphics.

APPENDIX A - USER COMMAND/METHOD DEFINITIONS

A. Turtle Class

1. forward

Description: Draws a line, in the direction of the input turtle heading, of length equal to the distance of the input number.

input: turtle; number (distance)

output: turtle

2. drawto

Description: Draws a line from the location of the input turtle to a specific location on the display screen.

input: turtle; point {X-vertical displacement Y-horizontal displacement}

output: turtle

3. goto

Description: Moves the input turtle from its present location to a specific location on the display screen. No line is drawn.

input: turtle; point {X-vertical displacement Y-horizontal displacement}

output: turtle

4. turnright

Description: Shifts the input turtle's heading, X-degrees, in a clockwise manner.

input: turtle; number (X-degrees)

output: turtle

5. turnleft

Description: Shifts the input turtle's heading, X-degrees, in a counterclockwise manner.

input: turtle; number (X-degrees)

output: turtle

6. turnto

Description: Shifts the input turtle's heading to a specific heading: North - 0 or 360, South - 180, East - 90, West - 270.

input: turtle; number (X-degrees)

output: turtle

7. penup

Description: Deactivates the drawing capability of the input turtle. Results of all commands after this command remain the same with the exception that no drawing will take place.

input: turtle

output: turtle

8. pendown

Description: Reactivates the drawing capability of the input turtle.

input: turtle

output: turtle

B. pTurtle Class

1. doSide&Turn

Description: Draws a line, in the direction of the input turtle heading, of length equal to the input number(distance). The turtle heading will then be updated by turning in the direction appropriate with the input number(degrees). A positive input number will yield a clockwise update, while a negative input number will cause a counterclockwise update. This command routine will be iterated as many times as the input integer.

input: turtle; number (distance); number (degrees); integer (iterations)

output: turtle

2. polygon

Description: Draws one or more (iterations) of a polygon whose side lengths are equal to the input number, and number of sides equal to the input integer. After each complete iteration, the initial turtle heading will be adjusted by adding an amount equal to (360 / number of iterations).

input: turtle; number (side length); integer (number of sides); integer (iterations)

output: turtle

3. square

Description: Draws one or more (iterations) of a square whose side lengths are equal to the input number. After each complete iteration, the initial turtle heading will be adjusted by adding an amount equal to (360 / number of iterations).

input: turtle; number (side length); integer(iterations)

output: turtle

4. triangle

Description: Draws one or more (iterations) of an equilateral triangle whose side length is equal to the input number. After each complete iteration, the initial turtle heading will be adjusted by adding an amount equal to (360 / number of iterations).

input: turtle; number (side length); integer(iterations)

output: turtle

5. circle

Description: Draws one or more (iterations) of a circle whose radius is equal to the input number. After each complete iteration, the turtle heading will be adjusted by adding an amount equal to (360 / number of iterations).

input: turtle; number (radius); number (radius); integer (iterations)

output: turtle

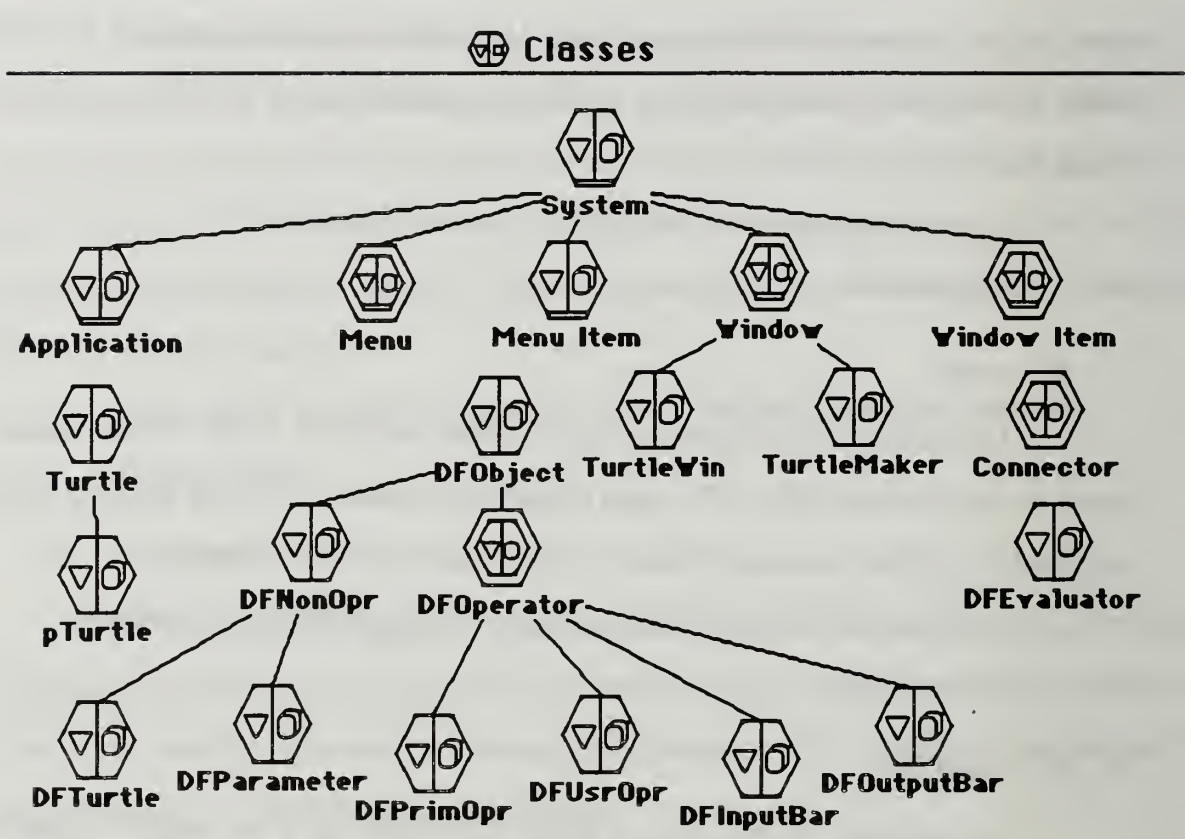
6. rectangle

Description: Draws one or more (iterations) of a rectangle whose side lengths are equal to the input numbers. After each complete iteration, the initial turtle heading will be adjusted by adding an amount equal to (360 / number of iterations).

input: turtle; number (side length); number (side length); integer (iterations)

output: turtle

APPENDIX B - NEW TURTLE GRAPHICS - SOURCE CODE



▽Turtle

NULL
▽
name

{ 200 200 }
▽
location

0
▽
heading

{ 1 1 }
▽
tailWidth

"Black"
▽
trailColor

TRUE
▽
Trail On?

Turtle

endown

input: turtle
output:turtle
Turn on turtle drawing ability.

penup

input: turtle
output:turtle
Turn off turtle drawing ability.

up to dra

input: turtle
output: none
Initialize pen characteristics.

rward

input: turtle, number (distance)
output:turtle
Draw number-length line in the forward direction.

goto

input: turtle, point{V H}
output:turtle
Moves turtle to specified location.

drawto

input: turtle, point{V H}
output:turtle
Draw line to specified point.

turnto

input: turtle, number(heading)
output:turtle
Update turtle heading to specified heading.

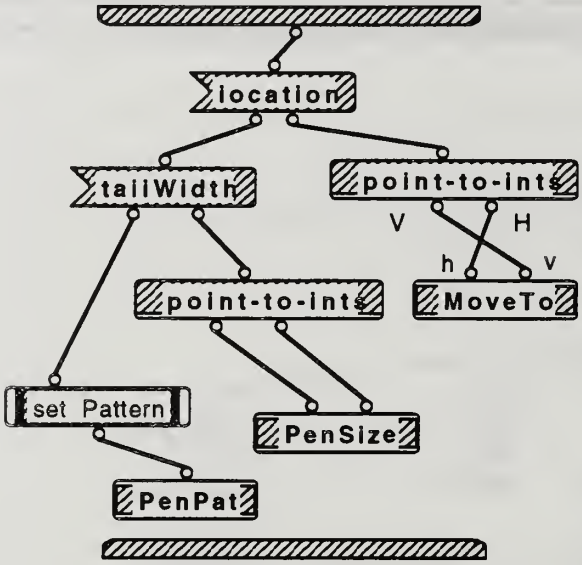
turnright

input: turtle, number(degrees)
output:turtle
Rotate turtle clockwise

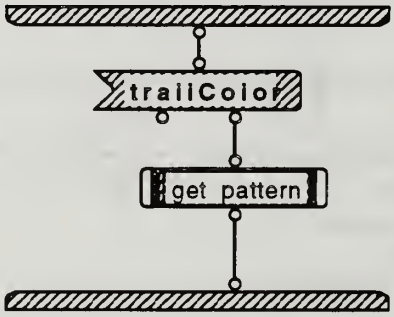
turnleft

input: turtle, number(degrees)
output:turtle
Rotate turtle counter-clockwise

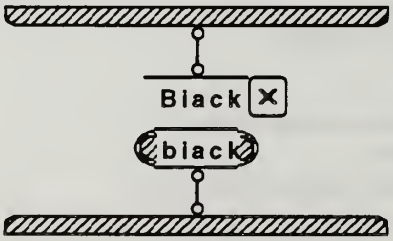
Turtle/set up to draw 1:1



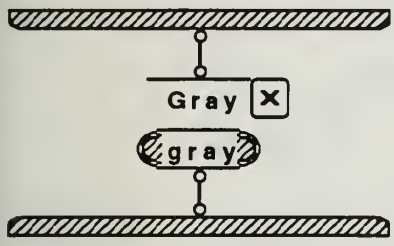
Turtle/set up to draw 1:1set Pattern 1:1



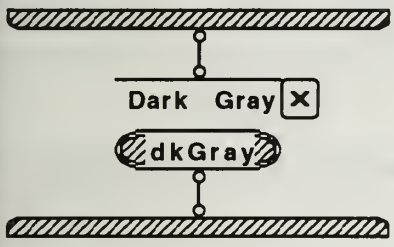
Turtle/set up to draw 1:1set Pattern 1:1get pattern 1:5



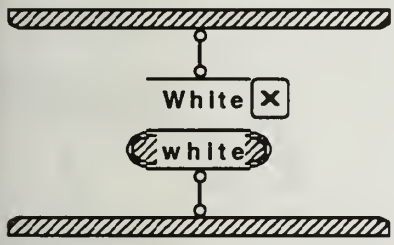
Turtle/set up to draw 1:1set Pattern 1:1get pattern 2:5



Turtle/set up to draw 1:1set Pattern 1:1get pattern 3:5

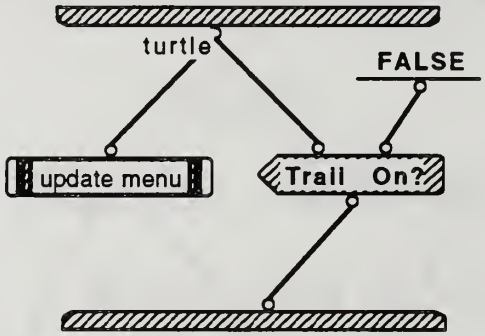


Turtle/set up to draw 1:1set Pattern 1:1get pattern 4:5

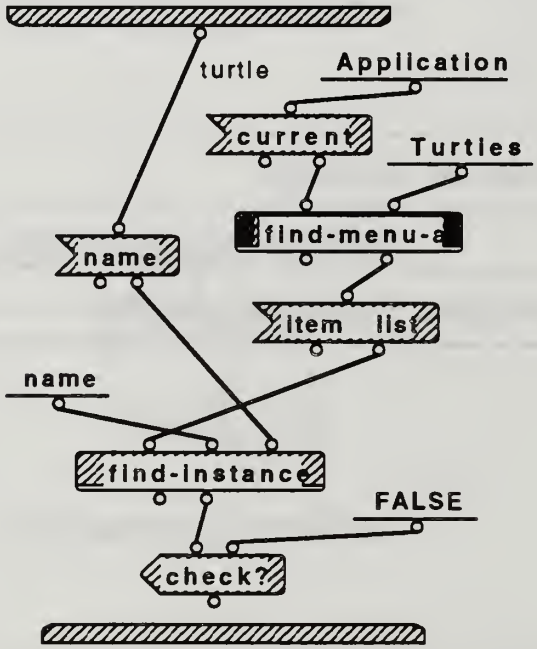


Turtle/set up to draw 1:1set Pattern 1:1get pattern 5:5

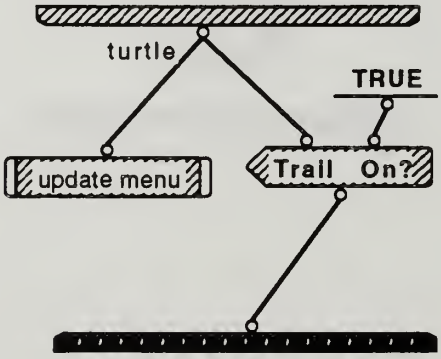
Turtle/penup 1:1



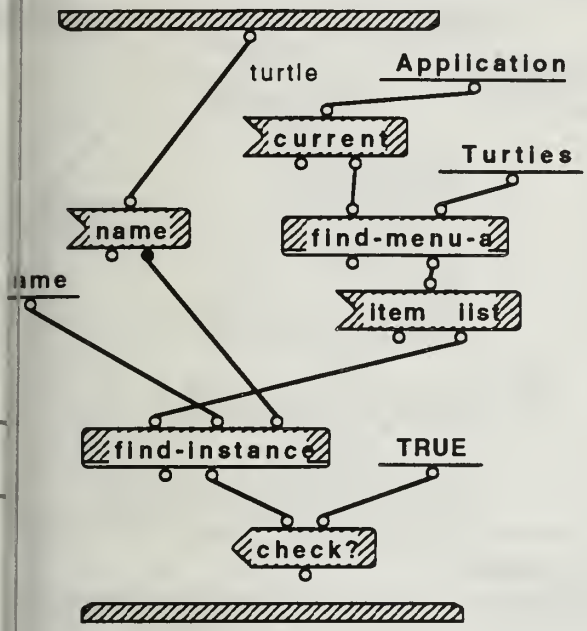
Turtle/penup 1:1update menu 1:1



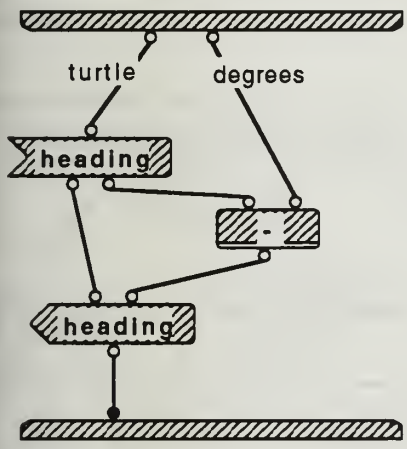
Turtle/pendown 1:1



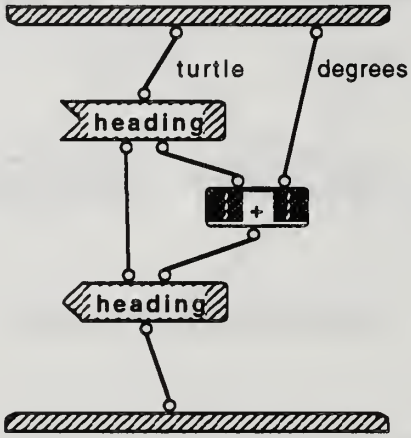
Turtle/pendown 1:1update menu 1:1



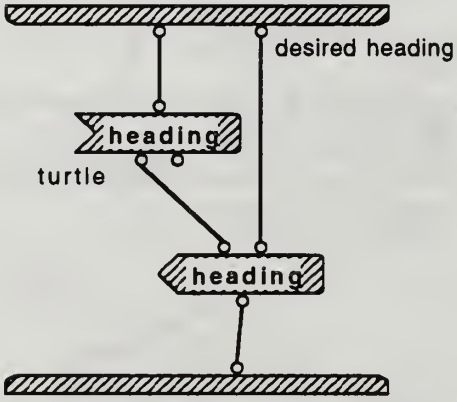
Turtle/turnleft 1:1



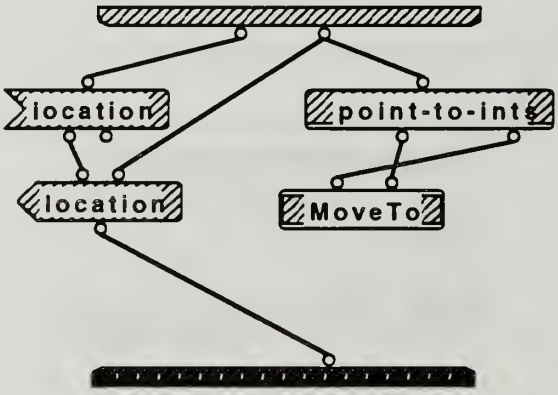
Turtle/turnright 1:1



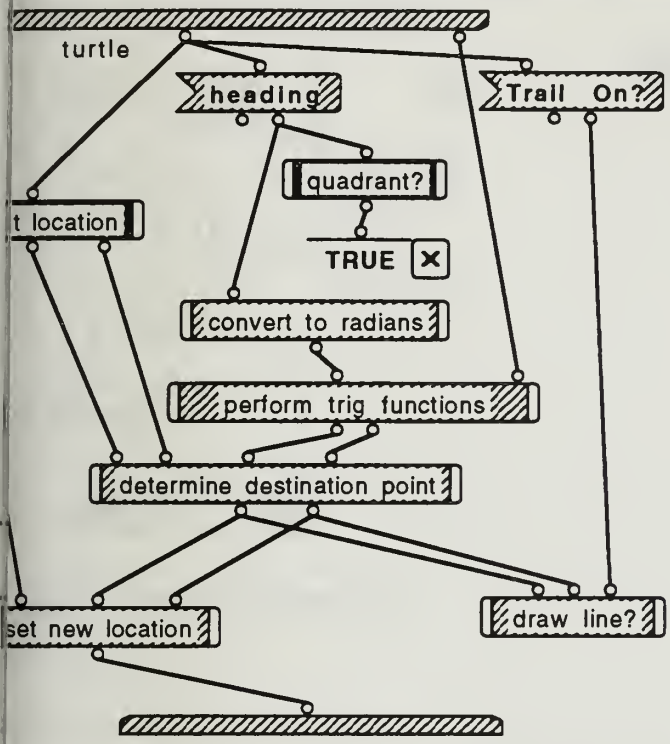
Turtle/turnto 1:1



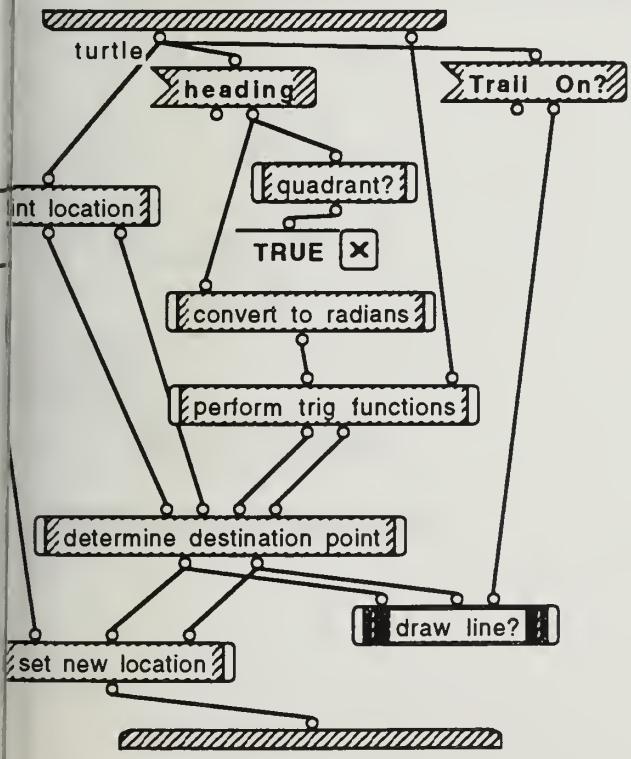
Turtle/goto 1:1



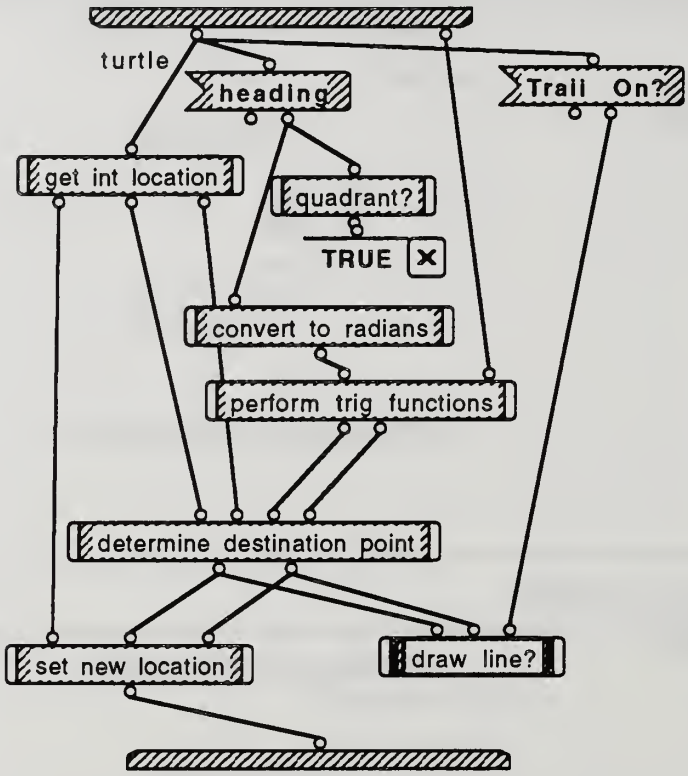
Turtle/forward 1:6



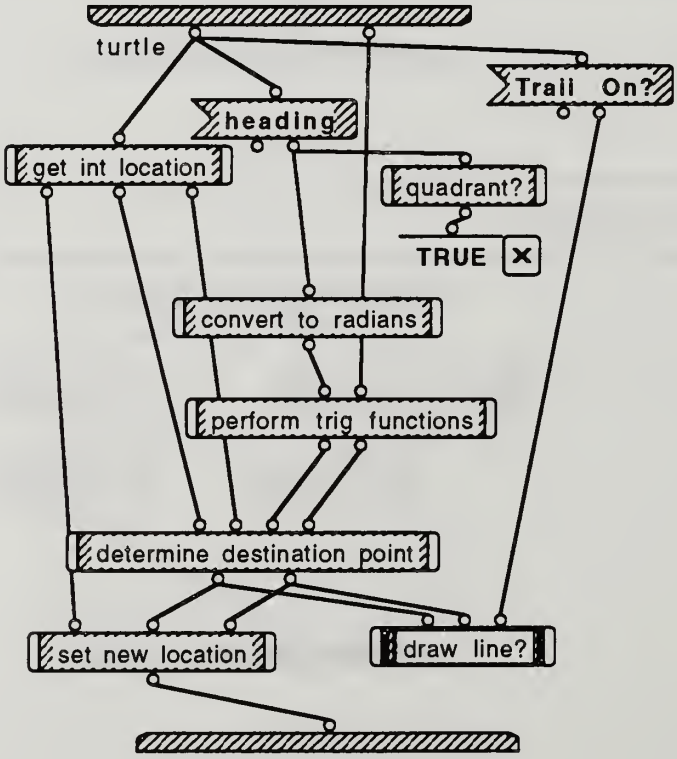
Turtle/forward 2:6



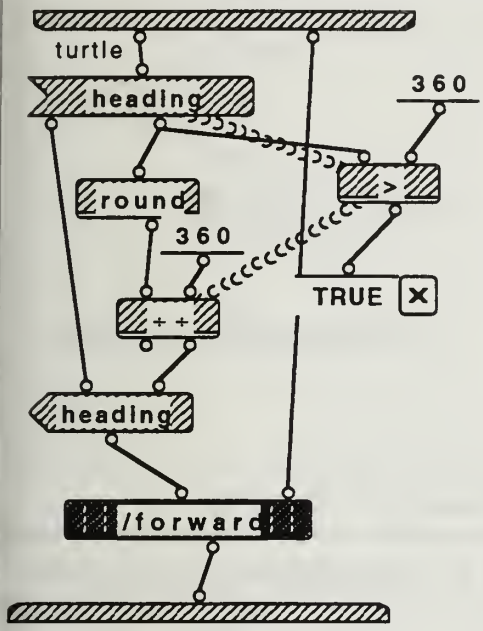
Turtle/forward 3:6



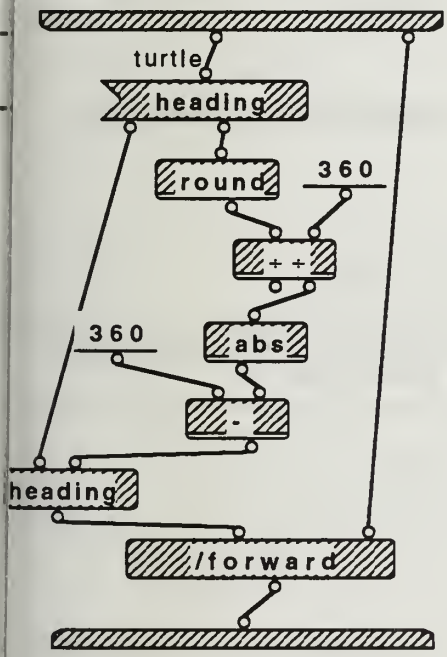
Turtle/forward 4:6



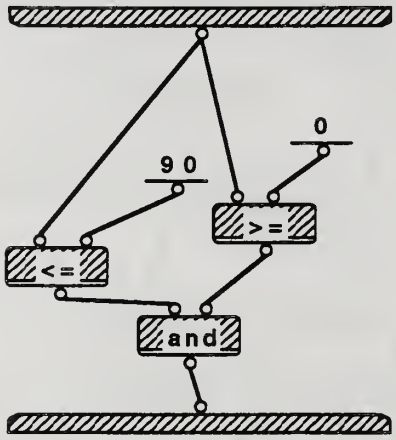
Turtle/forward 5:6



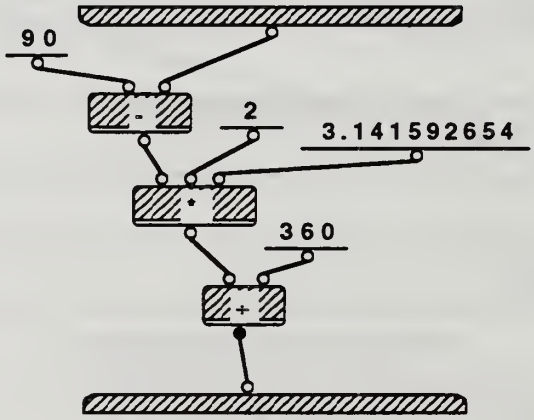
Turtle/forward 6:6



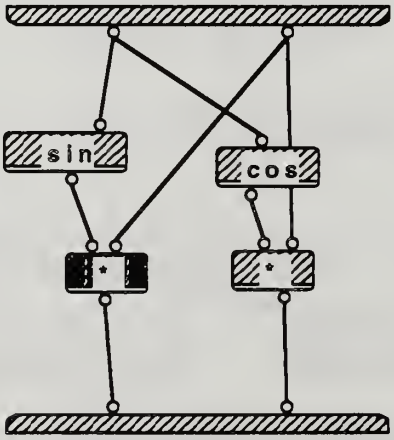
Turtle/forward 1:6quadrant? 1:1



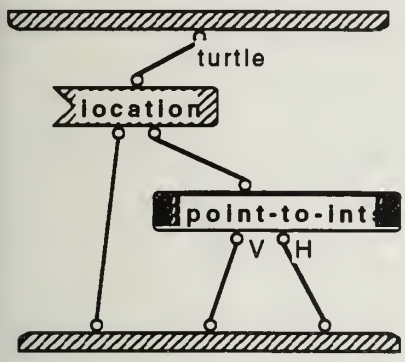
Turtle/forward 1:6convert to radians 1:1



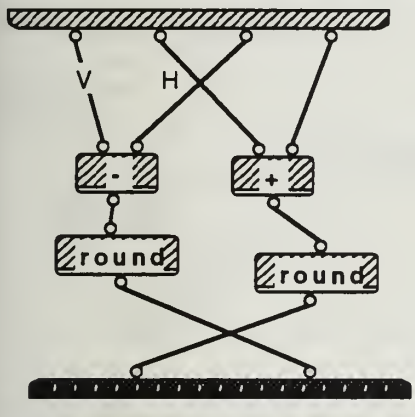
Turtle/forward 1:6perform trig functions 1:1



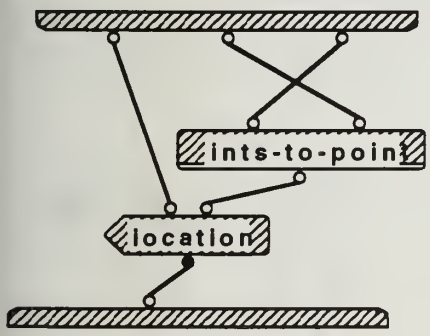
Turtle/forward 1:6get int location 1:1



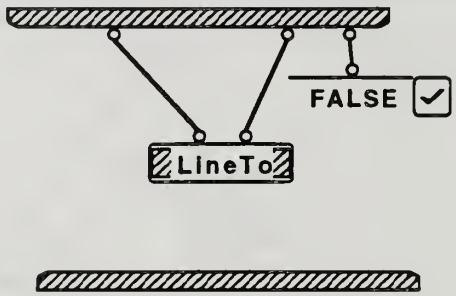
Turtle/forward 1:6determine destination point 1:1



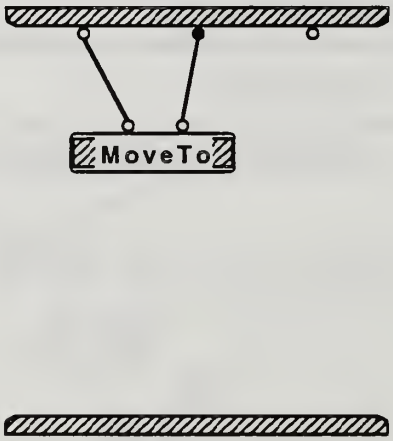
Turtle/forward 1:6set new location 1:1



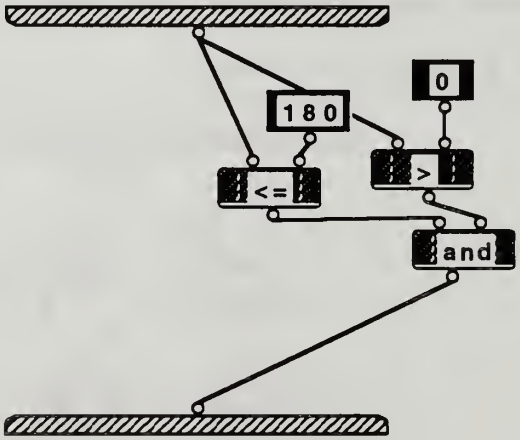
Turtle/forward 1:6draw line? 1:2



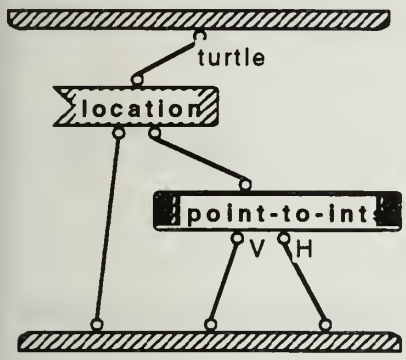
Turtle/forward 1:6draw line? 2:2



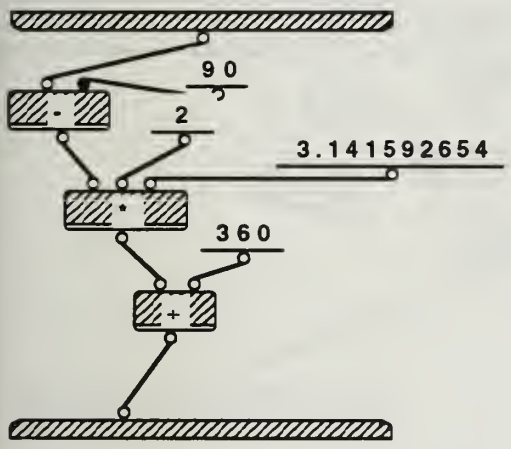
Turtle/forward 2:6quadrant? 1:1



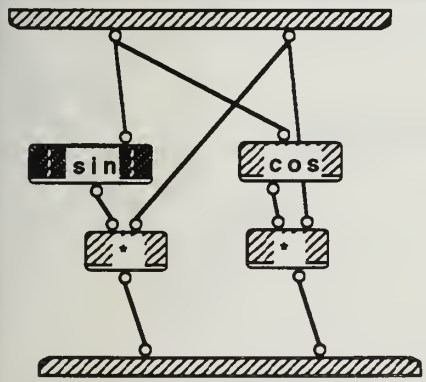
Turtle/forward 2:6get int location 1:1



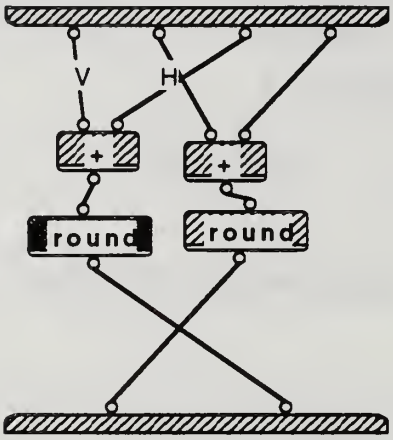
Turtle/forward 2:6convert to radians 1:1



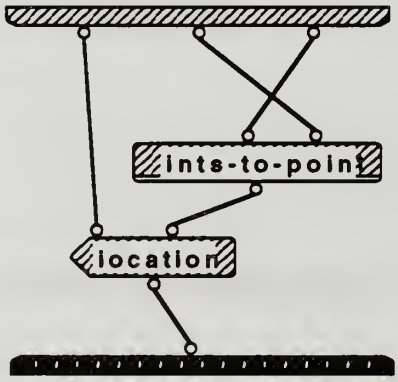
Turtle/forward 2:6perform trig functions 1:1



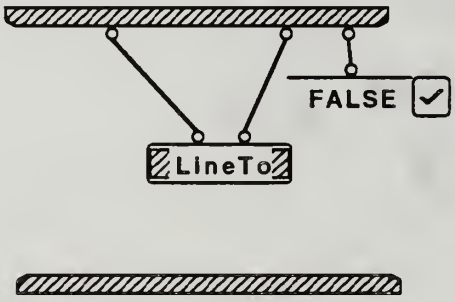
Turtle/forward 2:6determine destination point 1:1



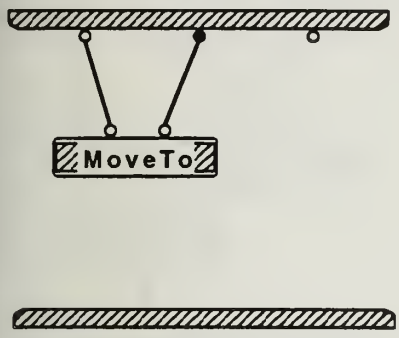
Turtle/forward 2:6set new location 1:1



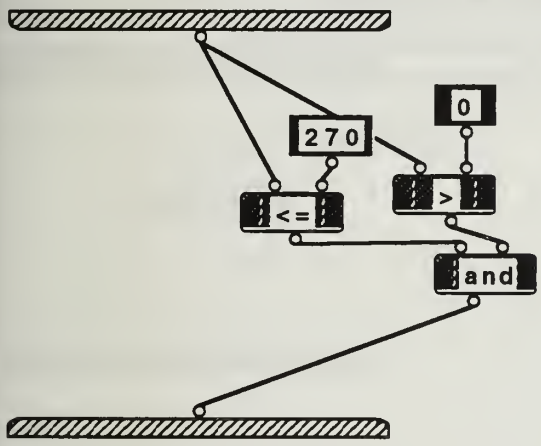
Turtle/forward 2:6draw line? 1:2



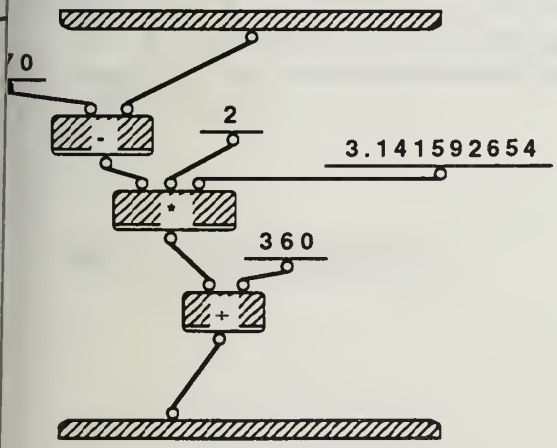
Turtle/forward 2:6draw line? 2:2



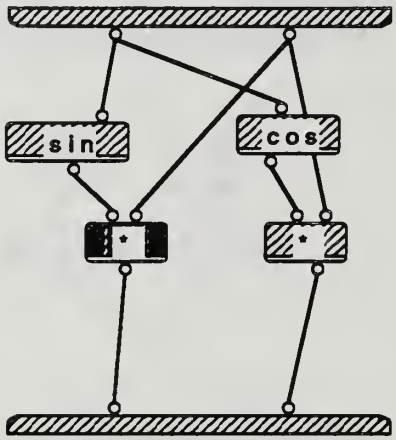
Turtle/forward 3:6quadrant? 1:1



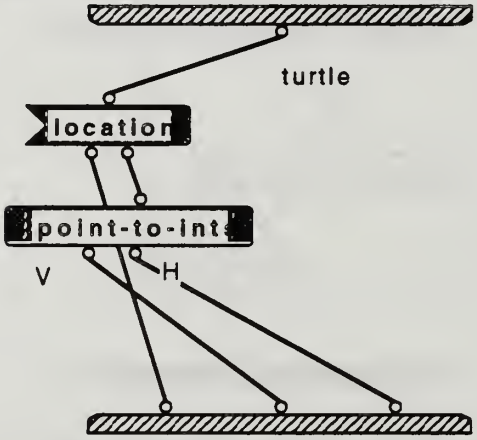
Turtle/forward 3:6convert to radians 1:1



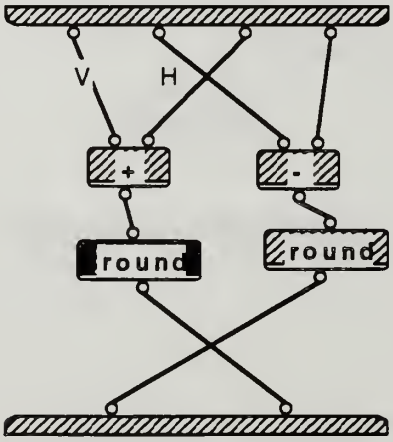
Turtle/forward 3:6perform trig functions 1:1



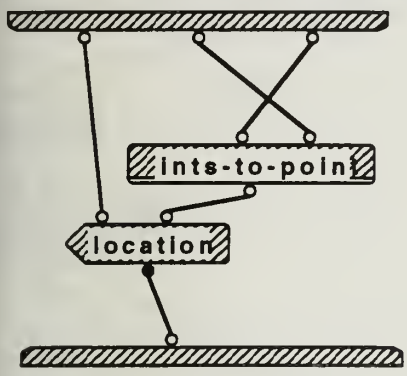
Turtle/forward 3:6get int location 1:1



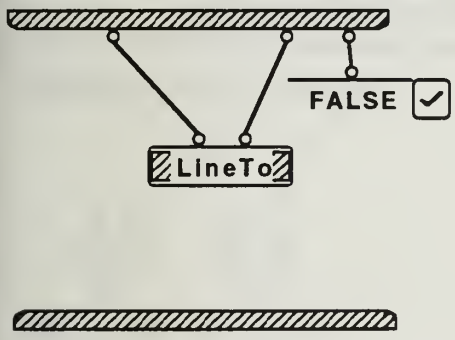
Turtle/forward 3:6determine destination point 1:1



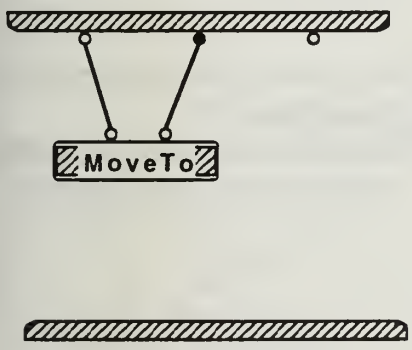
Turtle/forward 3:6set new location 1:1



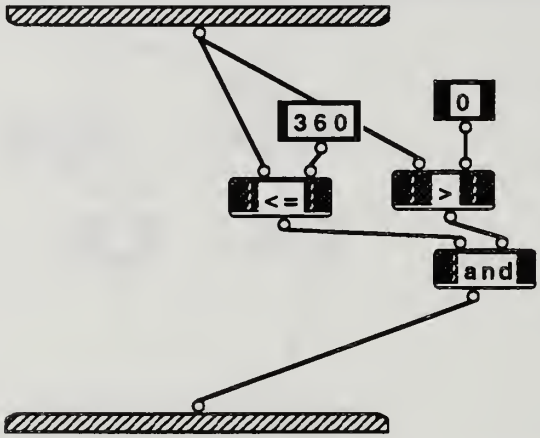
Turtle/forward 3:6draw line? 1:2



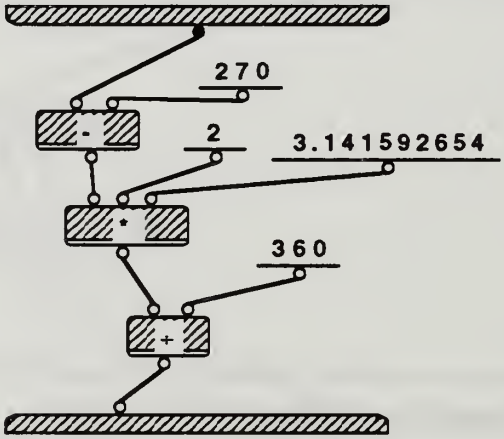
Turtle/forward 3:6draw line? 2:2



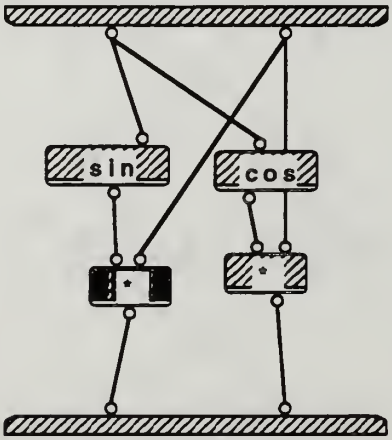
Turtle/forward 4:6quadrant? 1:1



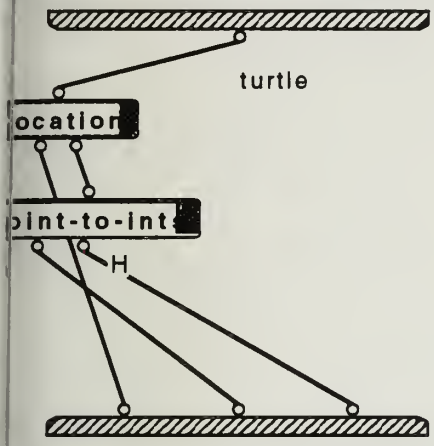
Turtle/forward 4:6convert to radians 1:1



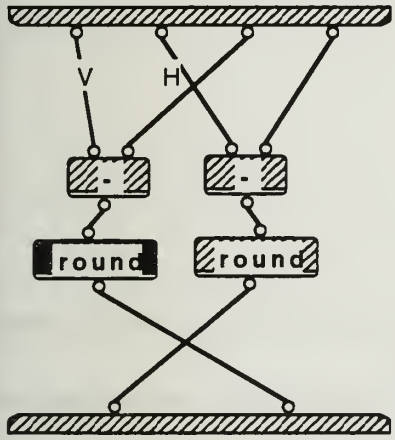
Turtle/forward 4:6perform trig functions 1:1



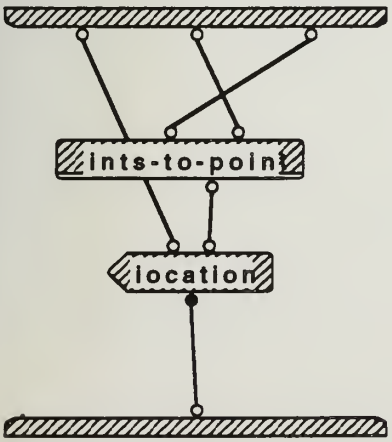
Turtle/forward 4:6get int location 1:1



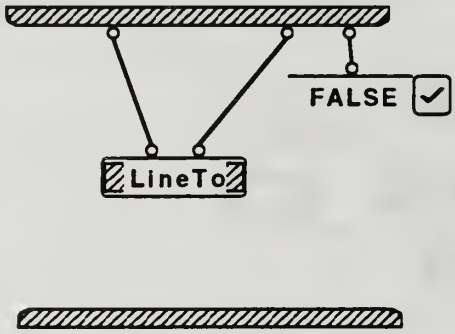
Turtle/forward 4:6determine destination point 1:1



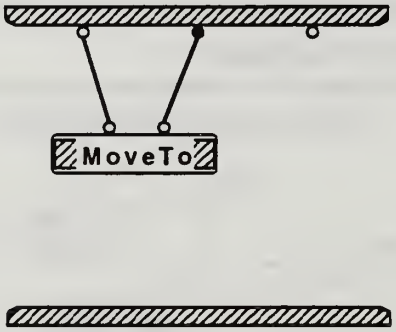
Turtle/forward 4:6set new location 1:1



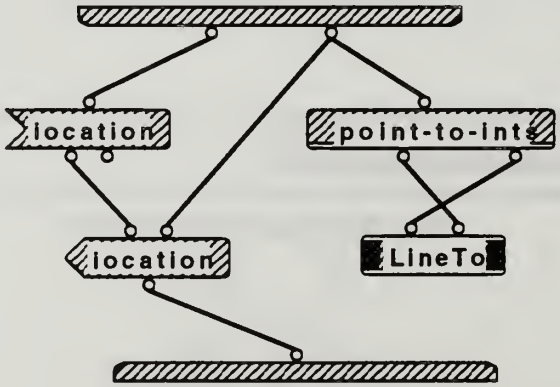
Turtle/forward 4:6draw line? 1:2



Turtle/forward 4:6draw line? 2:2



Turtle/drawto 1:1



▽ pTurtle

NULL
▽
name

NULL
▽
location

NULL
▽
heading

NULL
▽
tailWidth

NULL
▽
trailColor

NULL
▽
Trail On?

NULL
▽
program

NULL
▼
canvas pad

pTurtle

polygon input: turtle, number(side length), number(nr. of sides), number(nr. of polygons to draw)
output: turtle

circle input: turtle, number(radius), number(nr. of circles to draw)
output: turtle

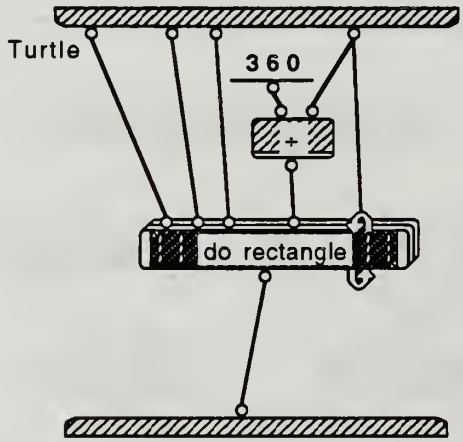
square input: turtle, number(side length), number(nr. of squares to draw)
output: turtle

rectangle input: turtle, number(side length), number(side length), number(nr. of rectangles to draw)
output: turtle

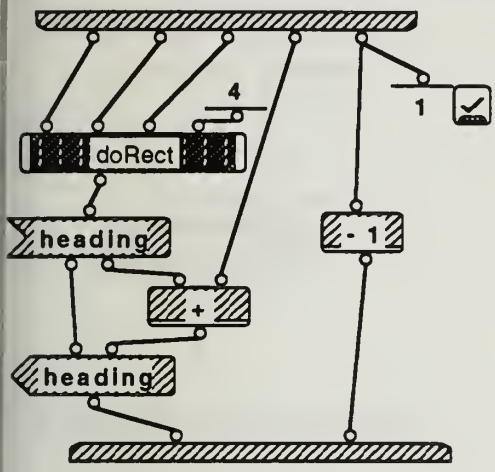
triangle input: turtle, number(side length), number(nr. of triangles to draw)
output: turtle

doSide&Turn input: turtle, number(side length), number(degrees to turn)
output: turtle

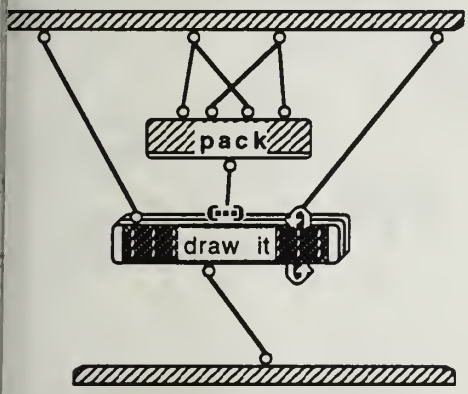
pTurtle/rectangle 1:1



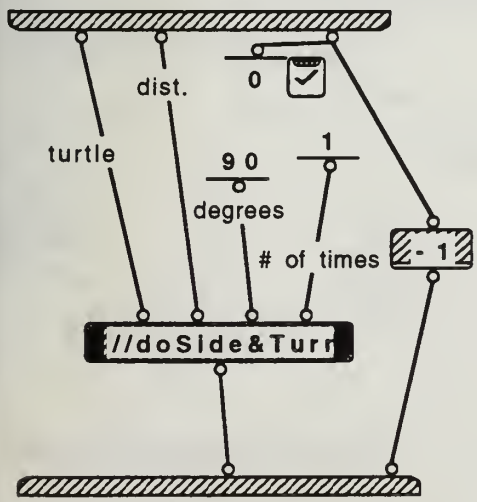
pTurtle/rectangle 1:1do rectangle 1:1



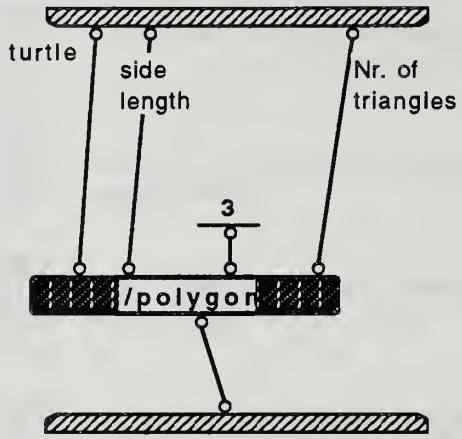
pTurtle/rectangle 1:1do rectangle 1:1doRect 1:1



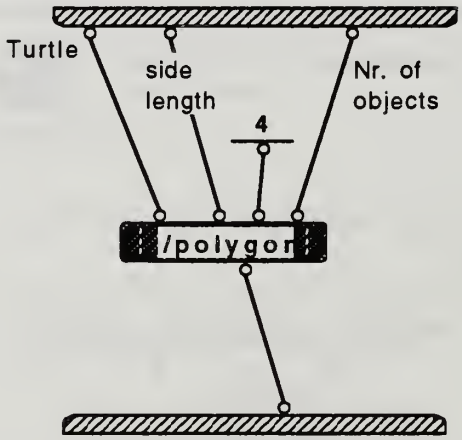
pTurtle/rectangle 1:1do rectangle 1:1doRect 1:1draw it 1:1



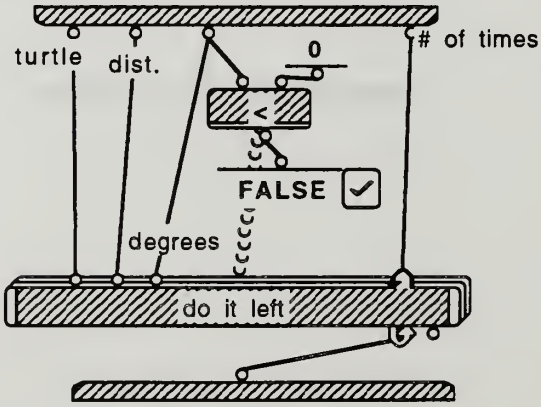
pTurtle/triangle 1:1



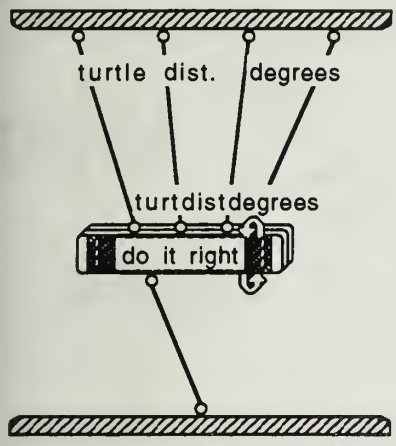
pTurtle/square 1:1



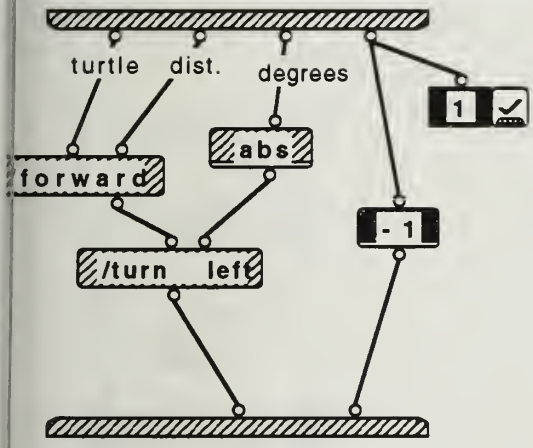
pTurtle/doSide&Turn 1:2



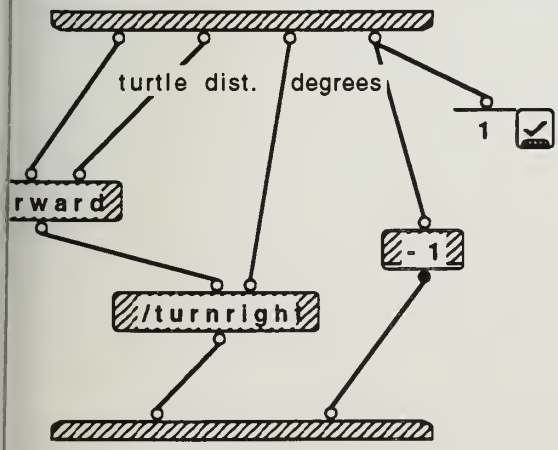
pTurtle/doSide&Turn 2:2



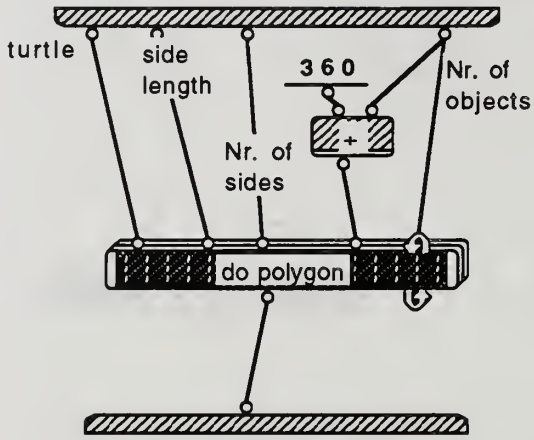
pTurtle/doSide&Turn 1:2do it left 1:1



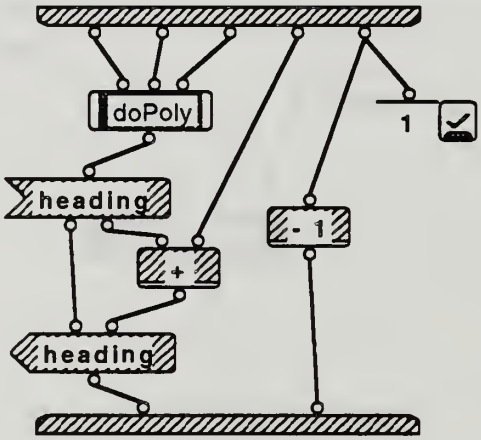
pTurtle/doSide&Turn 2:2do it right 1:1



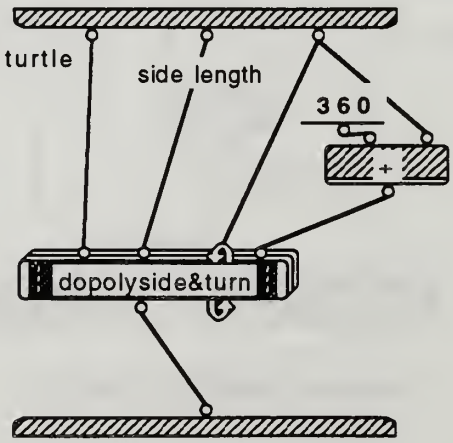
pTurtle/polygon 1:1



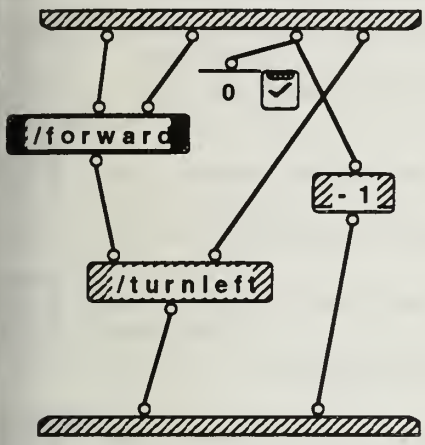
pTurtle/polygon 1:1do polygon 1:1



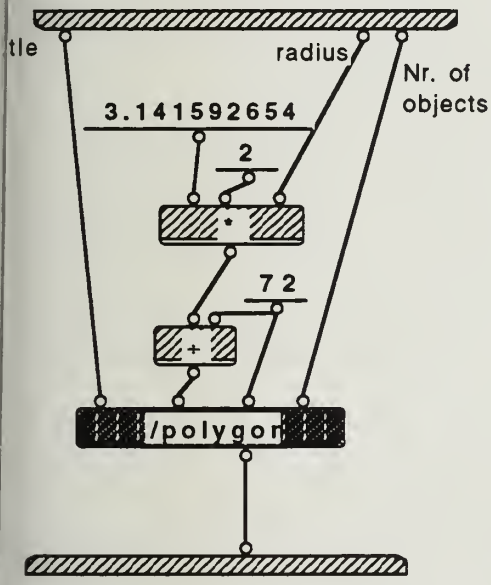
pTurtle/polygon 1:1do polygon 1:1doPoly 1:1



pTurtle/polygon 1:1do polygon 1:1doPoly 1:1dopolyside&turn 1:1



pTurtle/circle 1:1



"Turtle Disp...

name

NULL

owner

FALSE

active?

NULL

window record

8

def ID

FALSE

modal?

FALSE

close?

NULL

selected Item

{ 42 5 }

location

{ 250 250 }

size

..

activate method

..

close method

..

Idle method

..

key method

( )

Item list

( )

turtles

drawin

input: window, window item, event record
output: none
Prints the display graphics window.

turn

input: window, window item, event record
output: none
Returns to the Manipulation Window

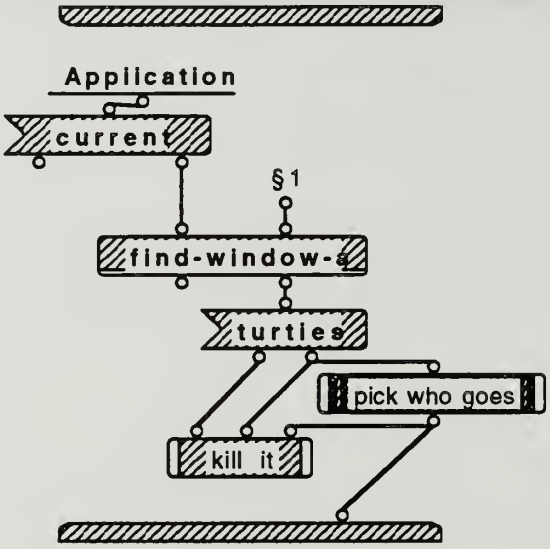
elete

input: window, window item, event record
output: none
Removes a turtle object from the menu.

edit

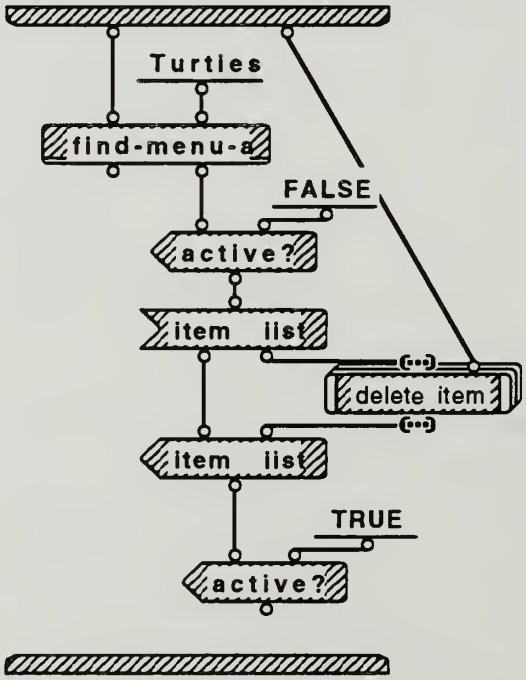
input: menu, menu item, event record
output: none
Opens the Turtle Definition window for editing

TurtleWin/Delete 1:1who goes? 1:1

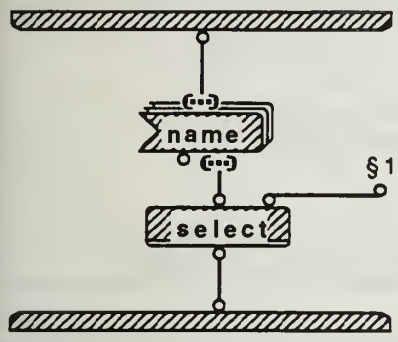


§1. Turtle Display

TurtleWin/Delete 1:1delete from menu 1:1

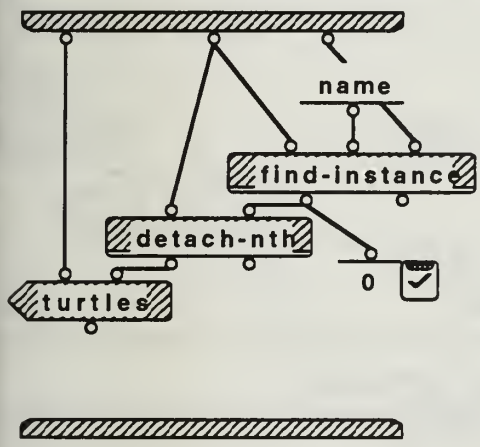


TurtleWin/Delete 1:1who goes? 1:1pick who goes 1:1

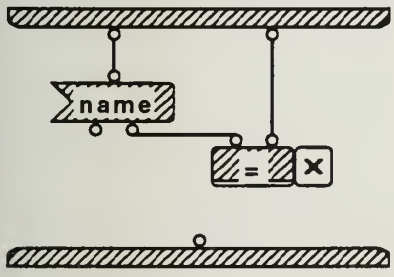


ch turtle do you want to delete?

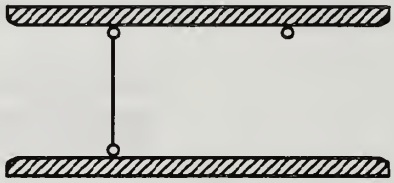
TurtleWin/Delete 1:1who goes? 1:1kill it 1:1



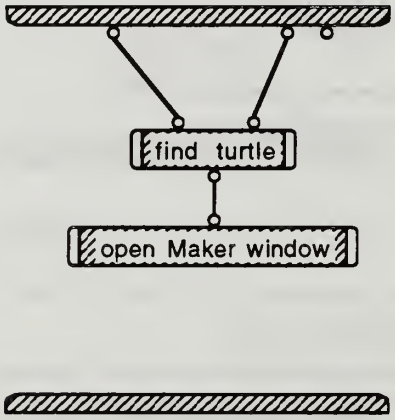
TurtleWin/Delete 1:1delete from menu 1:1delete item 1:2



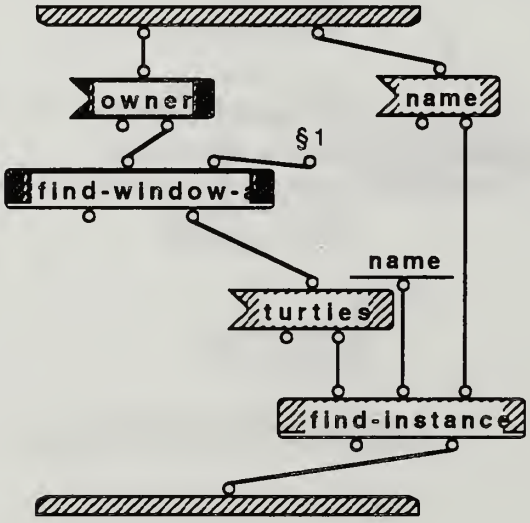
TurtleWin/Delete 1:1delete from menu 1:1delete item 2:2



TurtleWin/edit 1:1

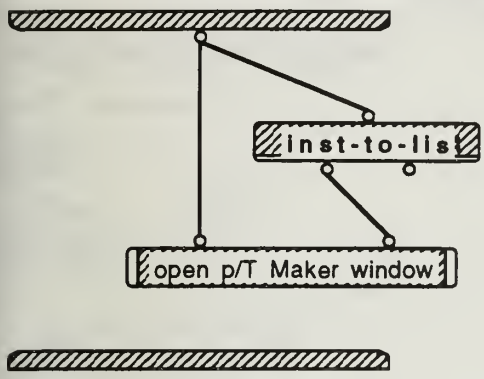


TurtleWin/edit 1:1find turtle 1:1

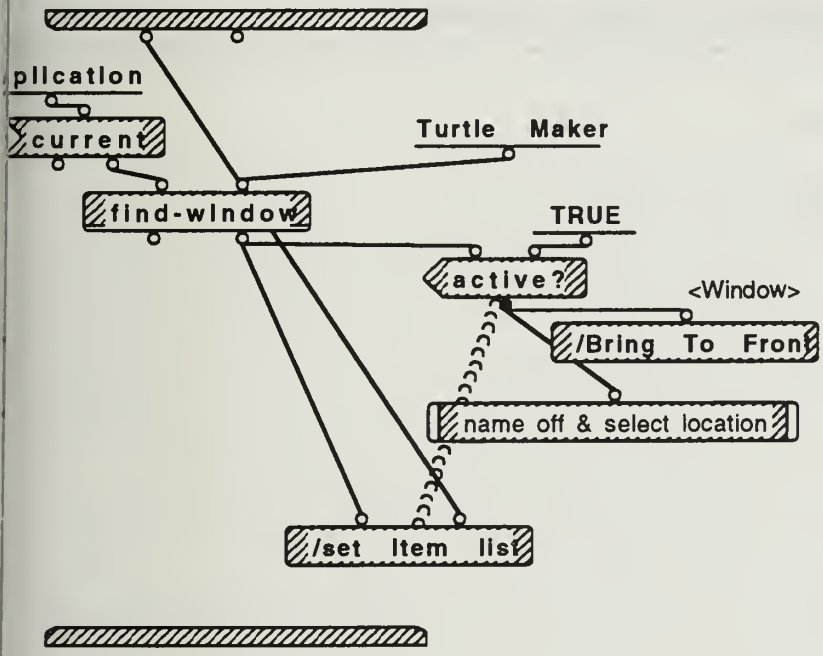


§1. Turtle Display

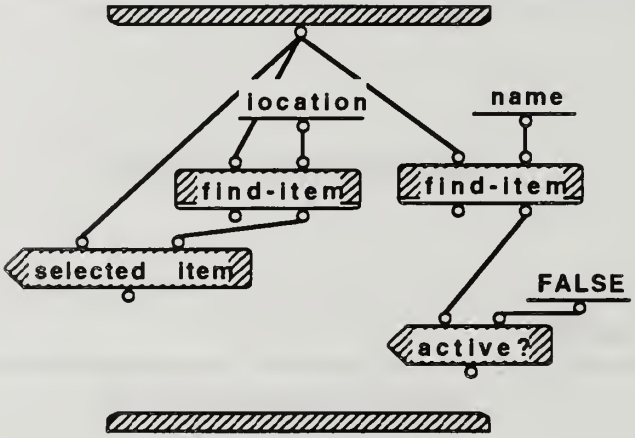
TurtleWin/edit 1:1open Maker window 1:1



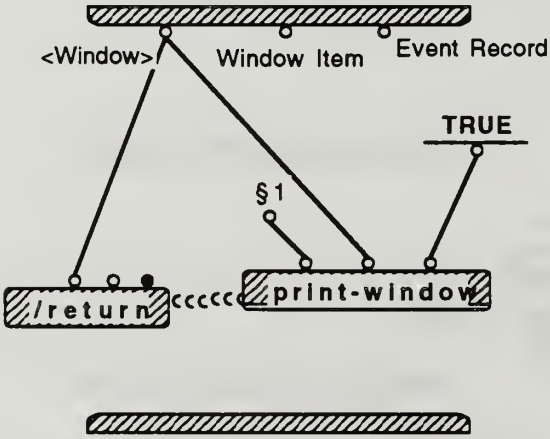
TurtleWin/edit 1:1open Maker window 1:1open p/T Maker window 1:1



TurtleWin/edit 1:1open Maker window 1:1open p/T Maker window 1:1name off & select location 1

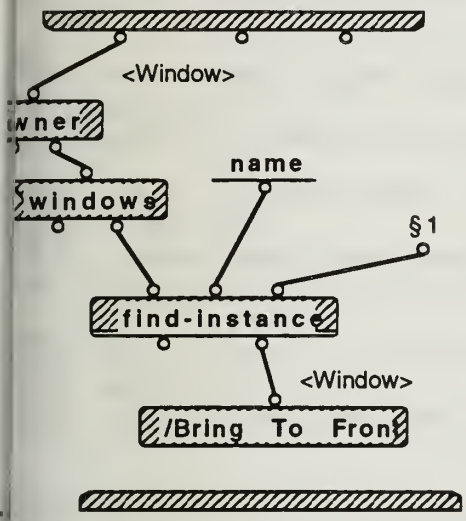


TurtleWin/print drawing 1:1



§1. Turtle Graphics Display Window!

TurtleWin/return 1:1



pulation Window

▽TurtleMaker

"Turtle Make...

name

NULL

owner

FALSE

active?

NULL

window record

4

def ID

FALSE

modal?

TRUE

close?

NULL

selected item

{ 40 106 }

location

{ 282 200 }

size

" "

activate method

"/close"

close method

" "

idle method

" "

key method

(<<Text>> <...

item list

TurtleMaker

K

input: TurtleMaker window, window item, event record
output: none
Initiates turtle creation for new turtles.

tem lls

input: TurtleMaker window, pTurtle
output: none
Sets the attribute values upon opening TurtleMaker window of defined turtle.

OKedIt

input: TurtleMaker window, window item, event record
output: none
Initiates turtle attribute definition update.

lose

input: TurtleMaker window, window item, event record
output: none
Brings main Manipulation window to front.

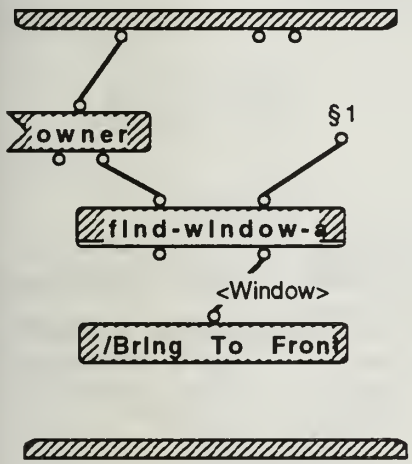
nMakei

input: menu, menu item, event record
output: none
Opens the TurtleMaker window

prep Attribute list

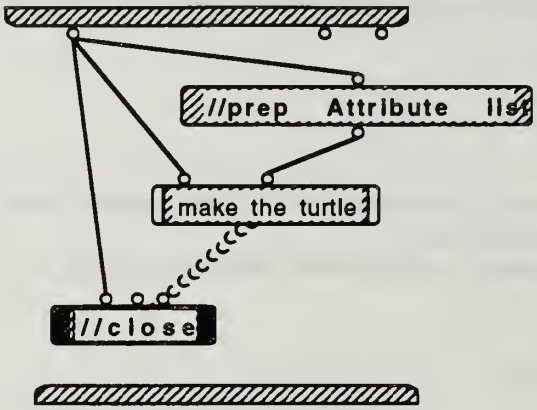
input: TurtleMaker window
output:
Establishes list of turtle attributes.

TurtleMaker/close 1:1

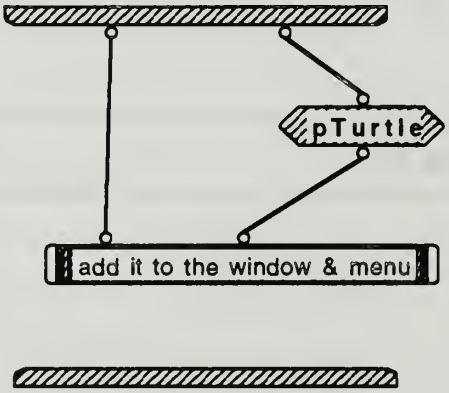


anipulation Window

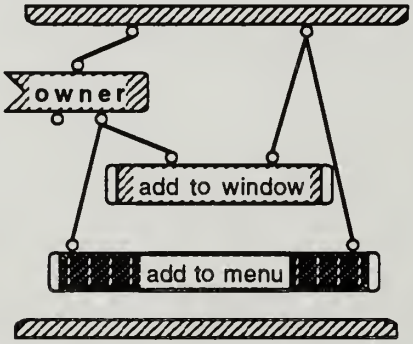
TurtleMaker/OK 1:1



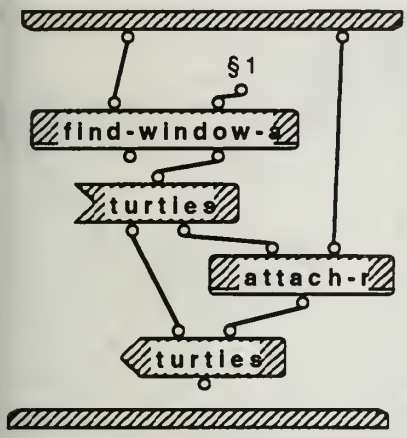
TurtleMaker/OK 1:1make the turtle 1:1



TurtleMaker/OK 1:1make the turtle 1:1add it to the window & menu 1

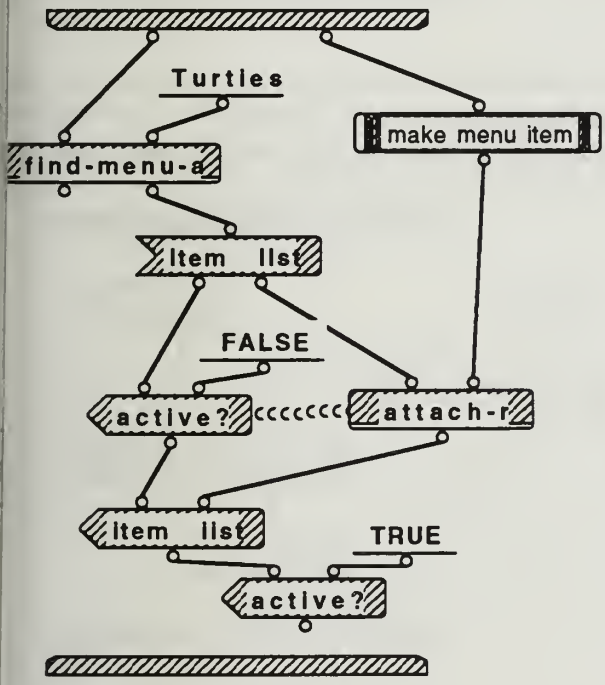


TurtleMaker/OK 1:1make the turtle 1:1add it to the window & menu 1:1add to window 1:1

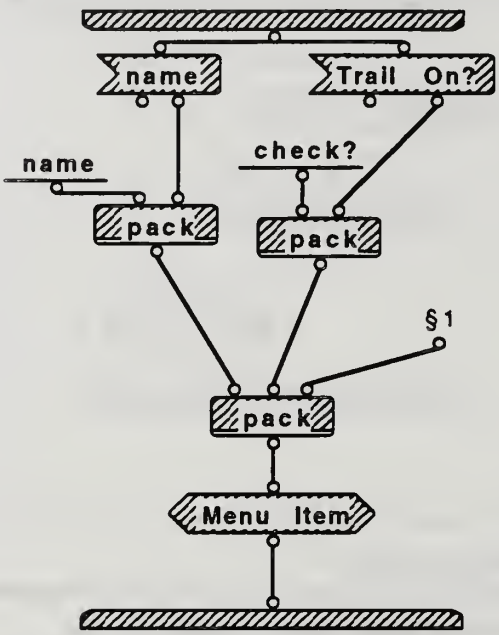


le Display

TurtleMaker/OK 1:1make the turtle 1:1add it to the window & menu 1:1add to menu 1:1

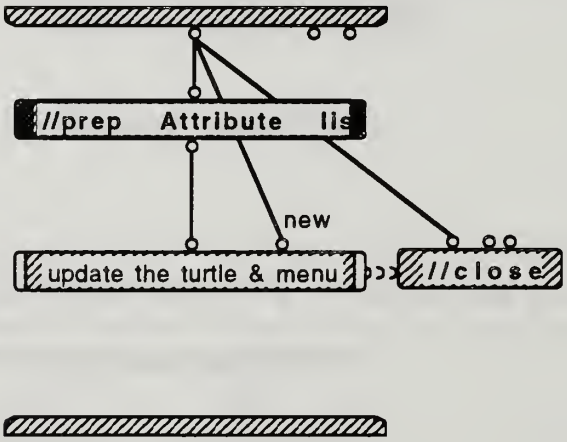


TurtleMaker/OK 1:1make the turtle 1:1add it to the window & menu 1:1add to menu 1:1make menu item

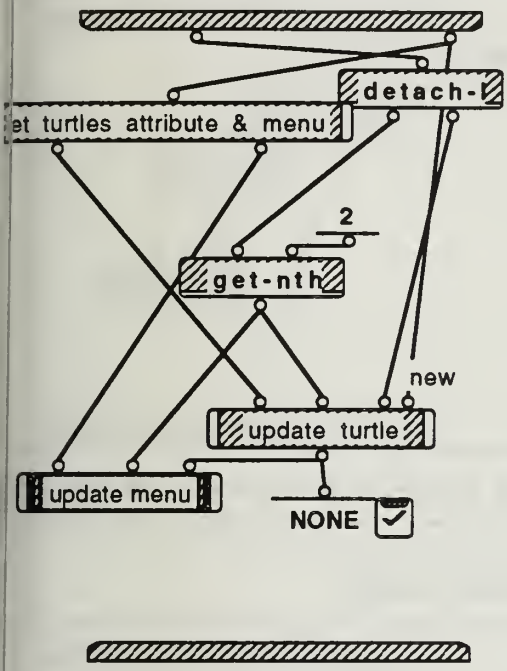


§1. (method "TurtleWin/edit")

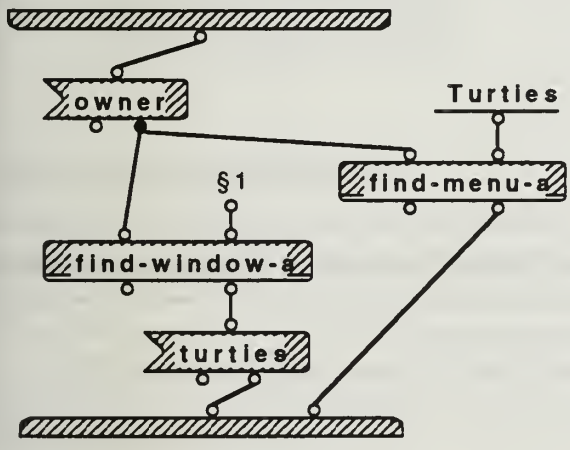
TurtleMaker/OKedit 1:1



TurtleMaker/OKedit 1:1update the turtle & menu 1:1

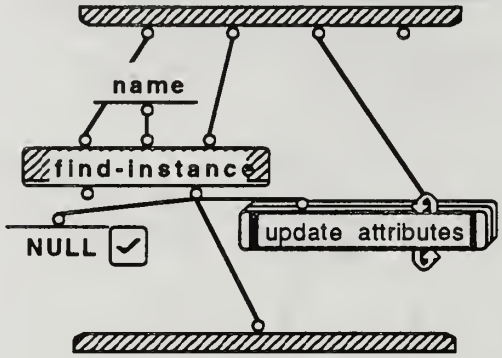


rtleMaker/OKedit 1:1update the turtle & menu 1:1get turtles attribute & menu 1:1

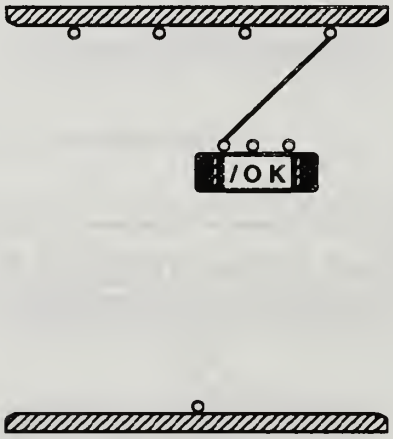


rtle Display

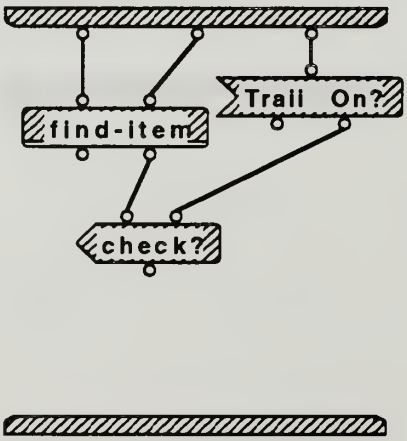
TurtleMaker/OKedit 1:1update the turtle & menu 1:1update turtle 1:



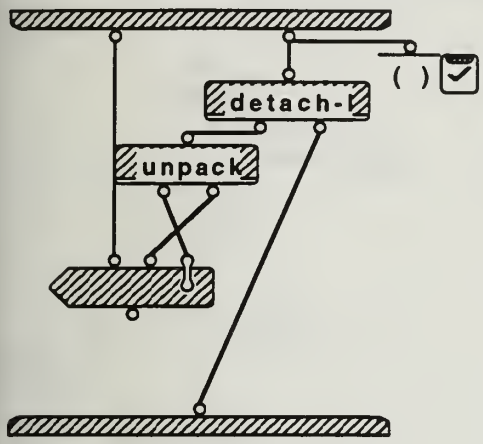
TurtleMaker/OKedit 1:1update the turtle & menu 1:1update turtle 2:



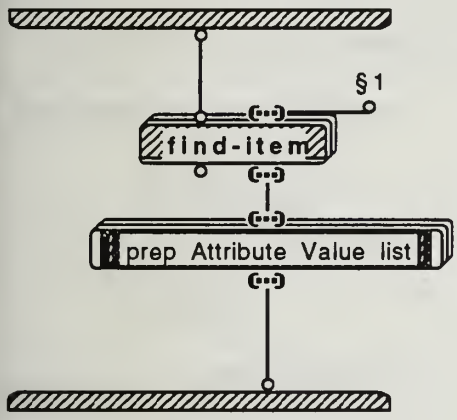
TurtleMaker/OKedit 1:1update the turtle & menu 1:1update menu 1:



TurtleMaker/OKedit 1:1update the turtle & menu 1:1update turtle 1:2update attributes 1:1

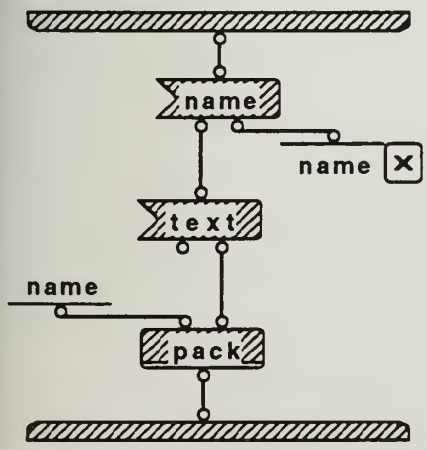


TurtleMaker/prep Attribute list 1:1

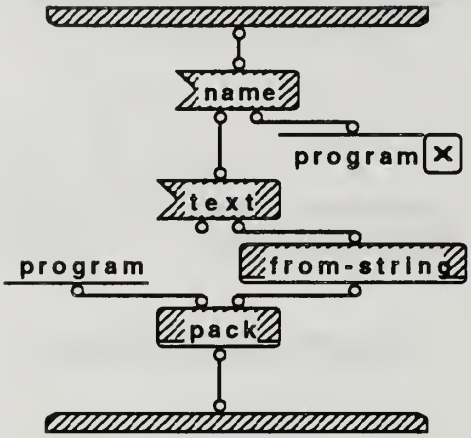


me location heading tailWidth trailColor "Trail On?")

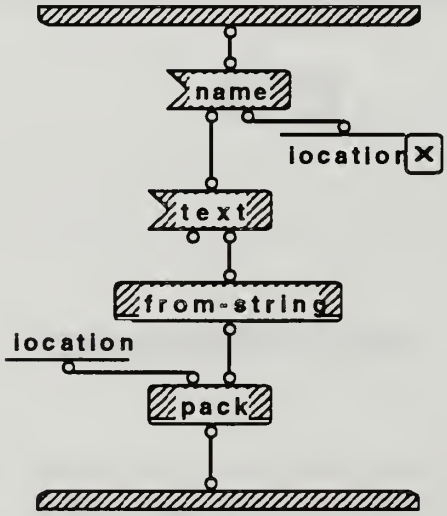
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 1:7



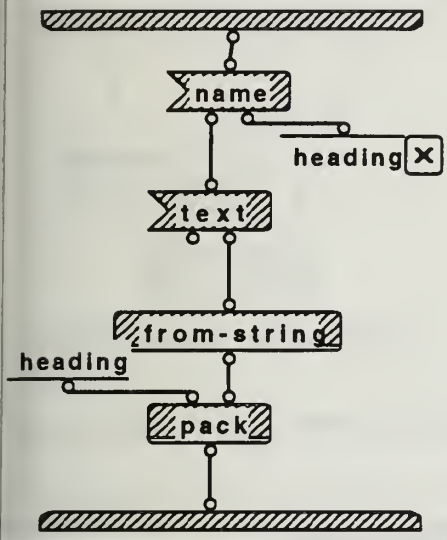
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 2:7



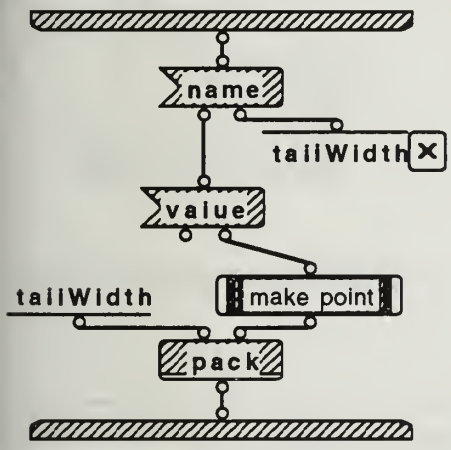
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 3:7



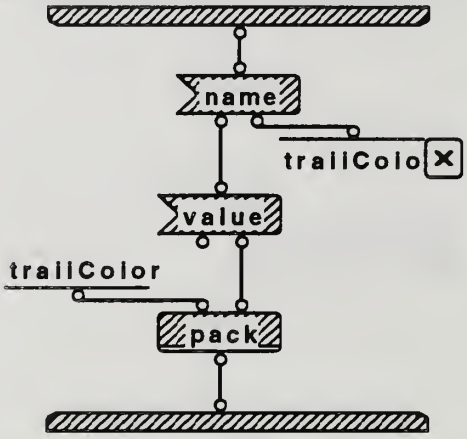
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 4:7



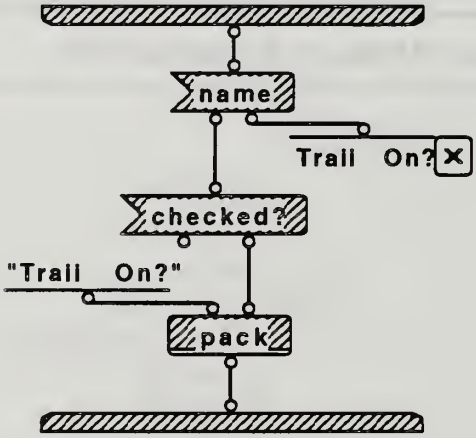
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 5:7



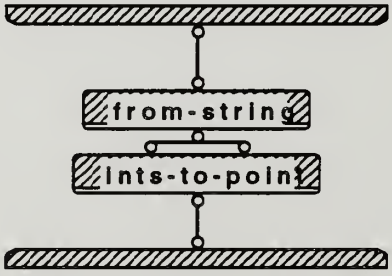
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 6:7



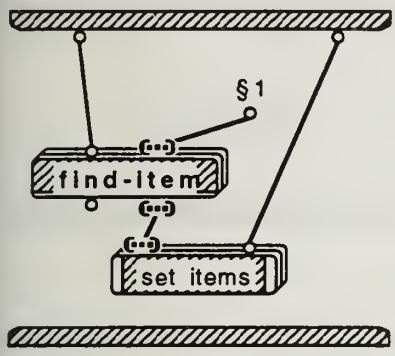
TurtleMaker/prep Attribute list 1:1prep Attribute Value list 7:7



TurtleMaker/prep Attribute list 1:1prep Attribute Value list 5:7make poin

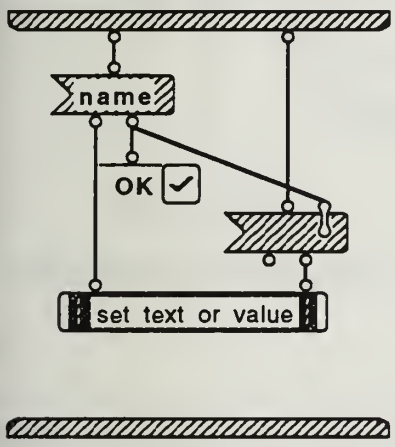


TurtleMaker/set item list 1:1

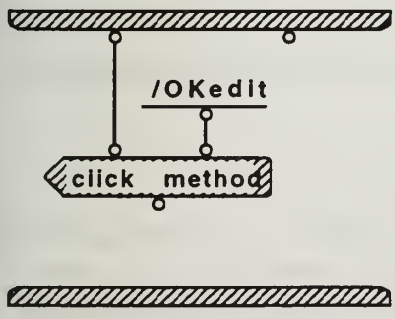


name location heading tailWidth trailColor "Trail On?")

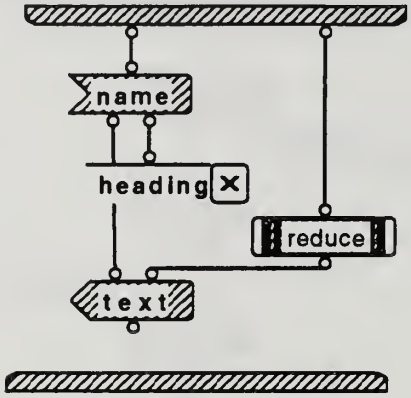
TurtleMaker/set item list 1:1set items 1:2



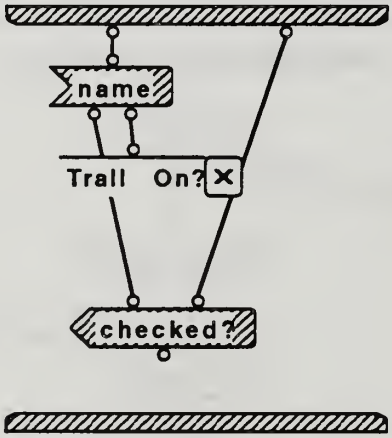
TurtleMaker/set item list 1:1set items 2:2



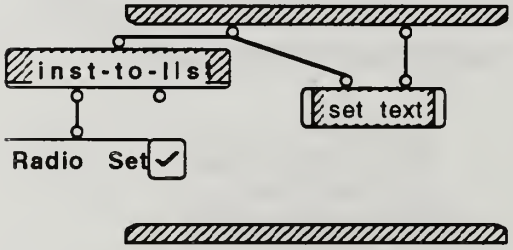
TurtleMaker/set item list 1:1set items 1:2set text or value 1:4



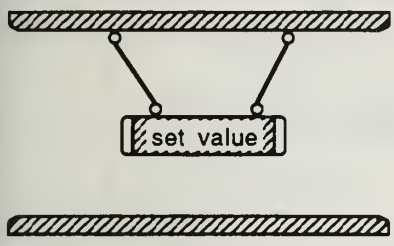
TurtleMaker/set item list 1:1set items 1:2set text or value 2:4



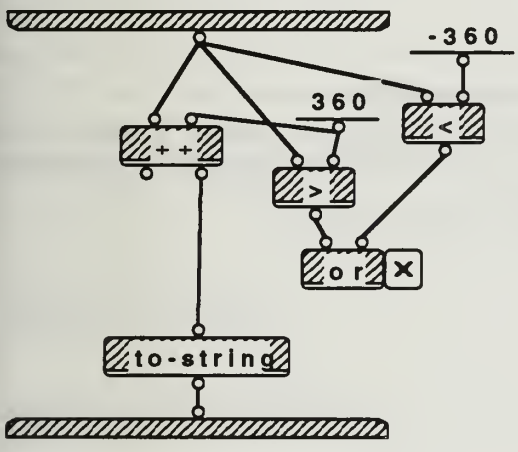
TurtleMaker/set item list 1:1set items 1:2set text or value 3:4



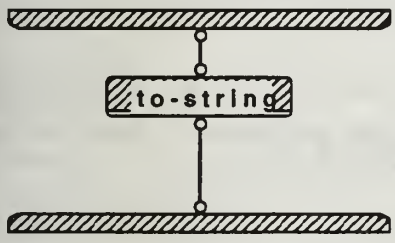
TurtleMaker/set item list 1:1set items 1:2set text or value 4:4



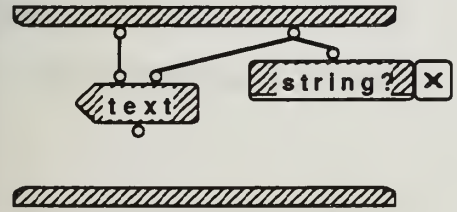
TurtleMaker/set item list 1:1set items 1:2set text or value 1:4reduce 1:2



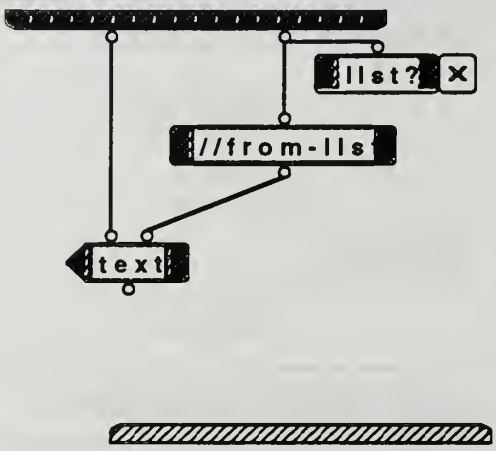
TurtleMaker/set item list 1:1set items 1:2set text or value 1:4reduce 2:2



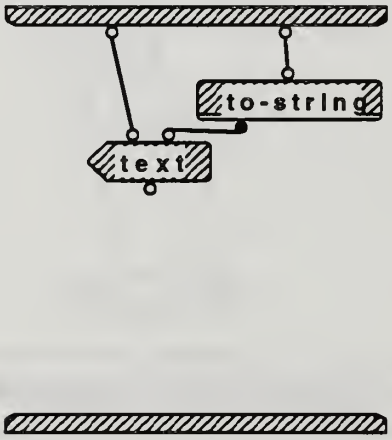
TurtleMaker/set item list 1:1set items 1:2set text or value 3:4set text 1:3



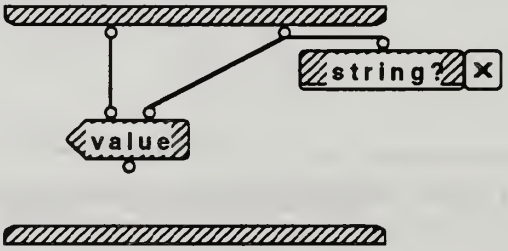
TurtleMaker/set item list 1:1set items 1:2set text or value 3:4set text



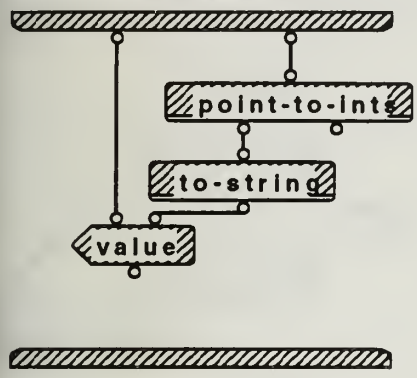
TurtleMaker/set item list 1:1set items 1:2set text or value 3:4set text



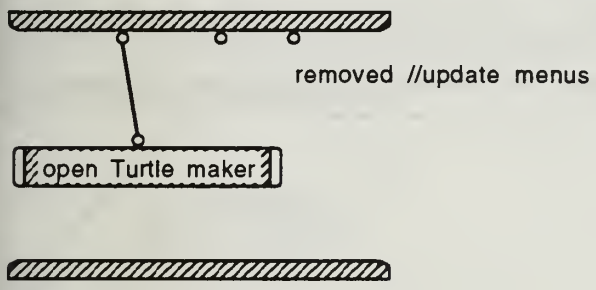
TurtleMaker/set item list 1:1set items 1:2set text or value 4:4set value



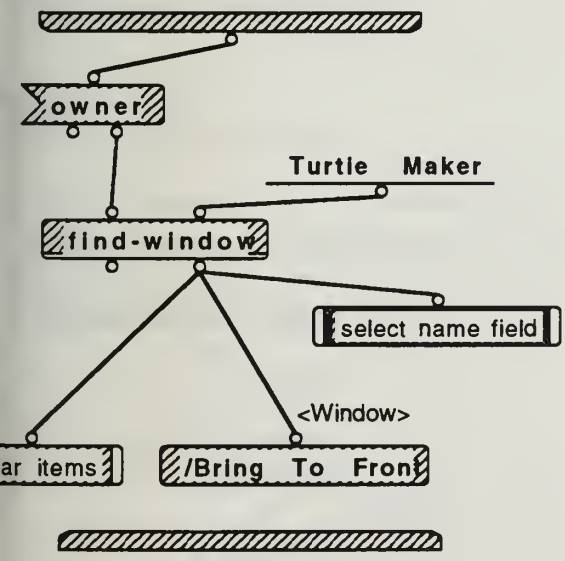
TurtleMaker/set item list 1:1set items 1:2set text or value 4:4set value 2:2



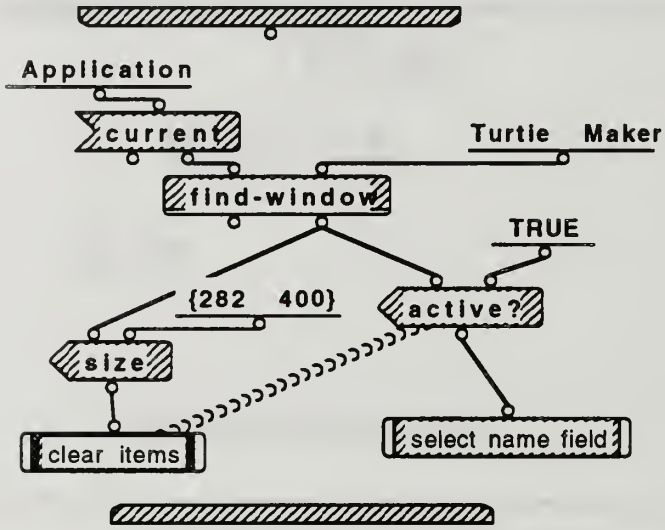
TurtleMaker/openMaker 1:1



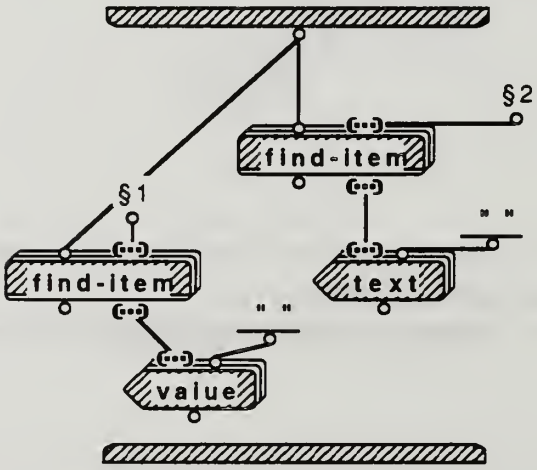
TurtleMaker/openMaker 1:1open Turtle maker 1:2



TurtleMaker/openMaker 1:1open Turtle maker 2:2

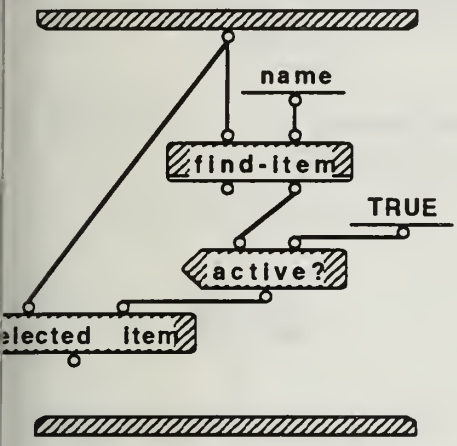


TurtleMaker/openMaker 1:1open Turtle maker 1:2clear items 1:1

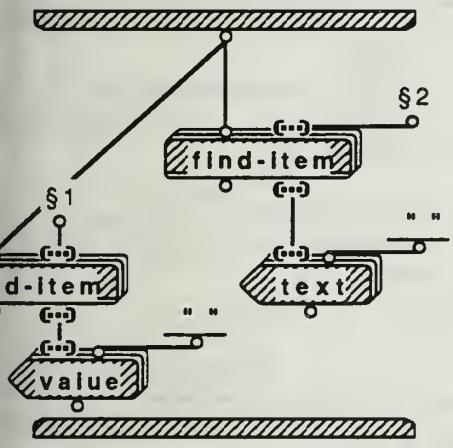


§1. (tailWidth trailColor)
§2. (name location heading)

TurtleMaker/openMaker 1:1open Turtle maker 1:2select name field 1:1

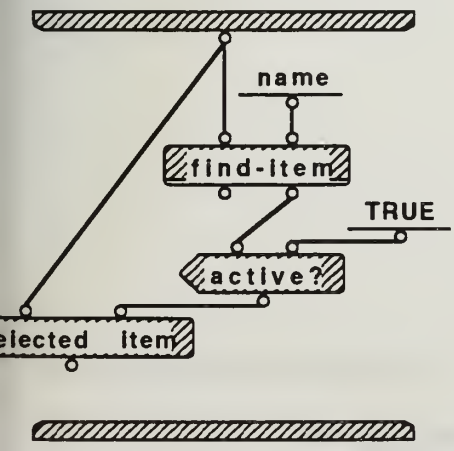


TurtleMaker/openMaker 1:1open Turtle maker 2:2clear items 1:1



Width trailColor)
ne program location heading)

TurtleMaker/openMaker 1:1open Turtle maker 2:2select name field 1:1



▽DFObject

NULL <Root> object

▽

root

{ 0 0 20 0 } location of its body

▽

bodyRect

NULL

▽

rootValue

FALSE

▽

selected?

DFObject

get root — private: create <Root> and attach it

move to — move it to the new location

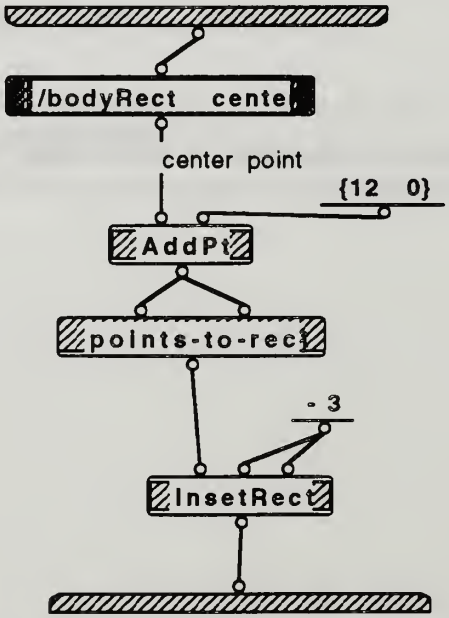
toggle — select/deselect it

bodyRect center — private: computer center of its body rect

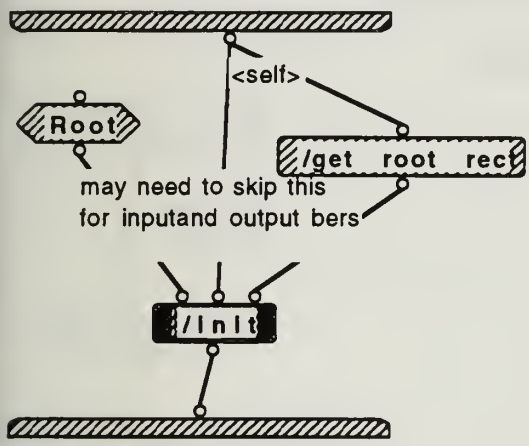
Invert

get root rect — return rectangle coordinates for root

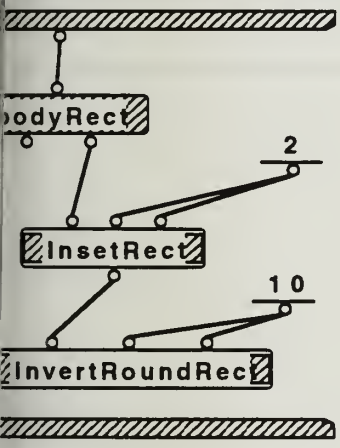
DFObject/get root rect 1:1



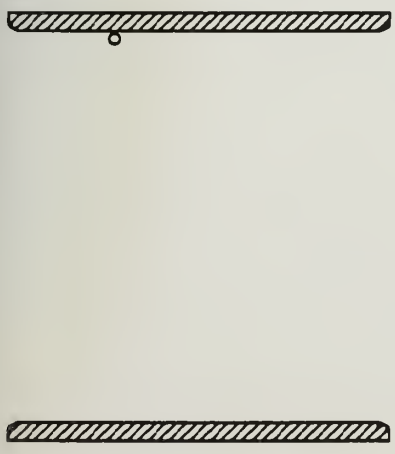
DFObject/get root 1:1



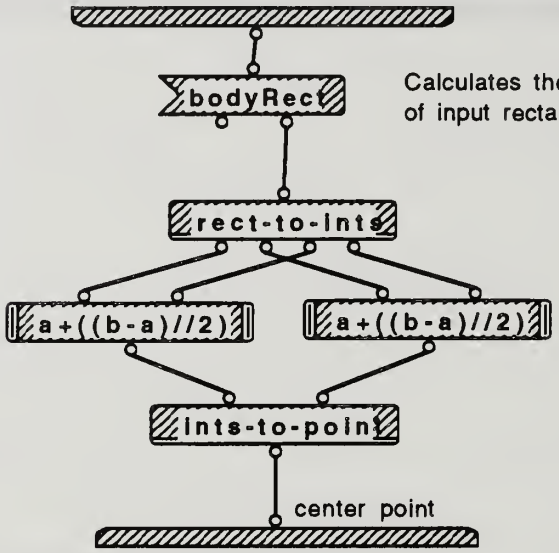
DFObject/invert 1:2



DFObject/invert 2:2

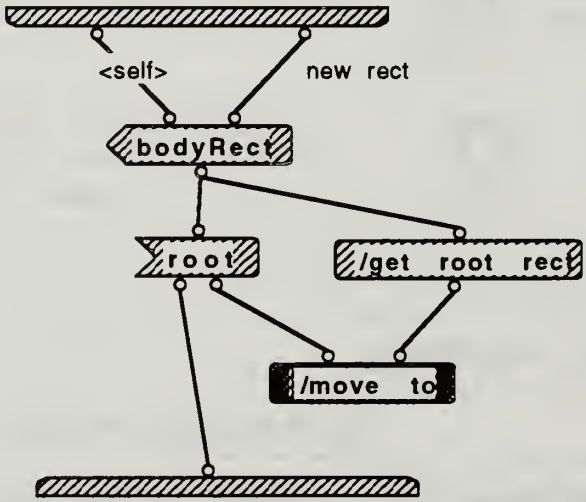


DFObject/bodyRect center 1:1

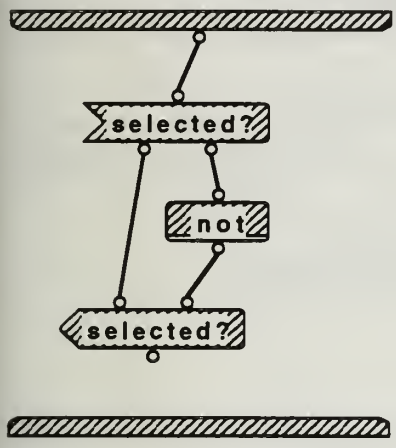


Calculates the center point (horiz) of input rectangle.

DFObject/move to 1:1



DFObject/toggle 1:1



▽ DFDperator

NULL <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

oprname

( ) list of <Terminal>
--((termrect fromObjInstnum) ..)

terminals

DFOperator

get terminals — returns terminals connected to object

init — init itself

get terminal rects — private: return a list of rects for its terminals

remove show info — displays info on itself

move to — allows objects to be around in window

get terminals cnt — private: return the number of terminals for itself

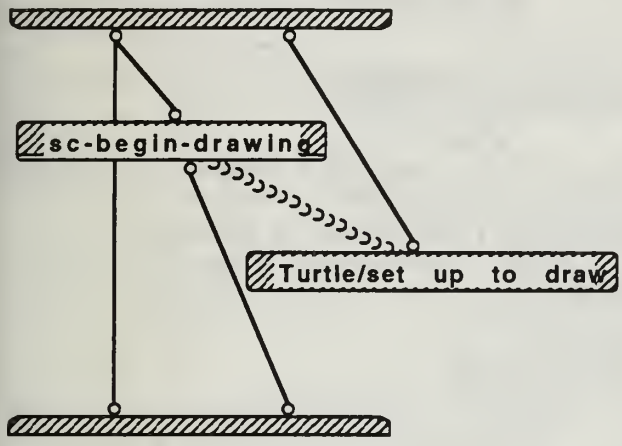
draw — draw itself on the currently "sc-begin" canvas

init draw — Initialize canvas and pen characteristics.

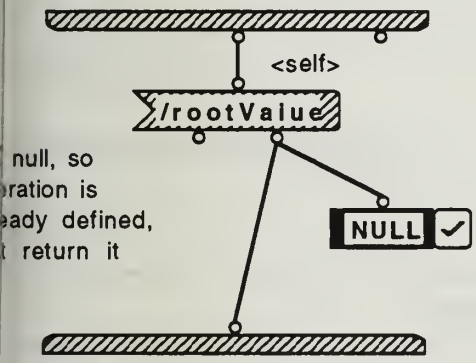
end draw — End drawing routine.

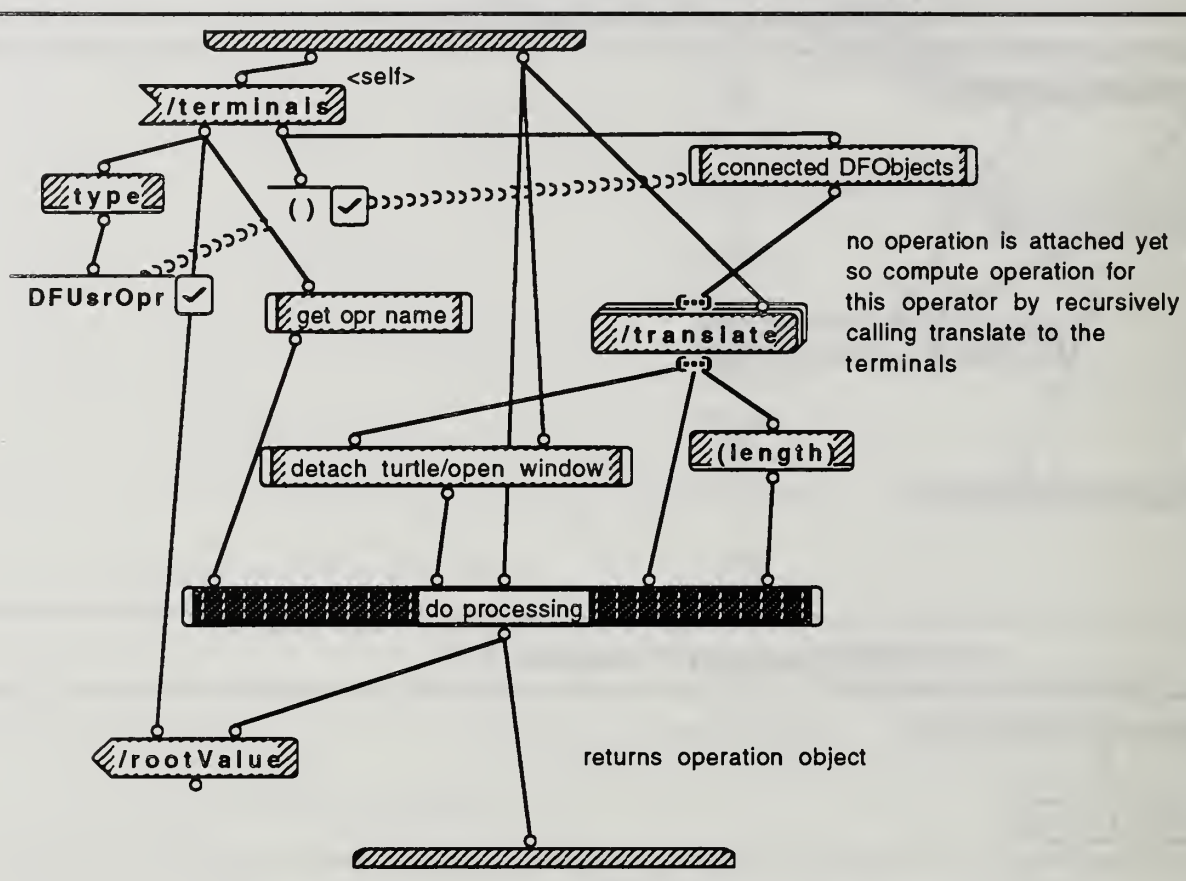
translate — translates/executes program

DFOperator/init draw 1:1

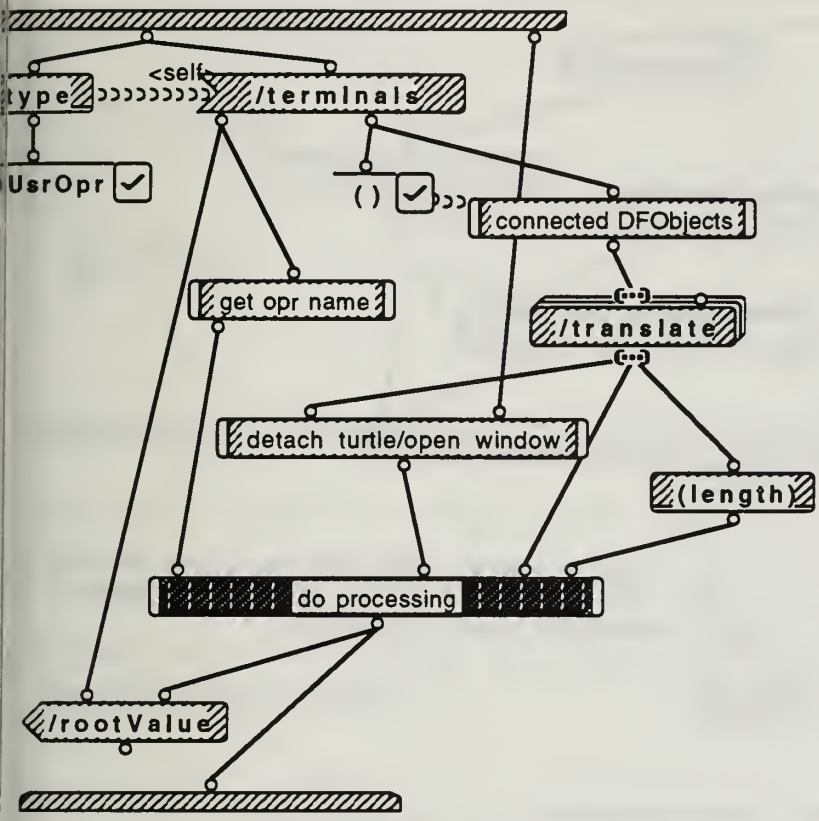


DFOperator/translate 1:5

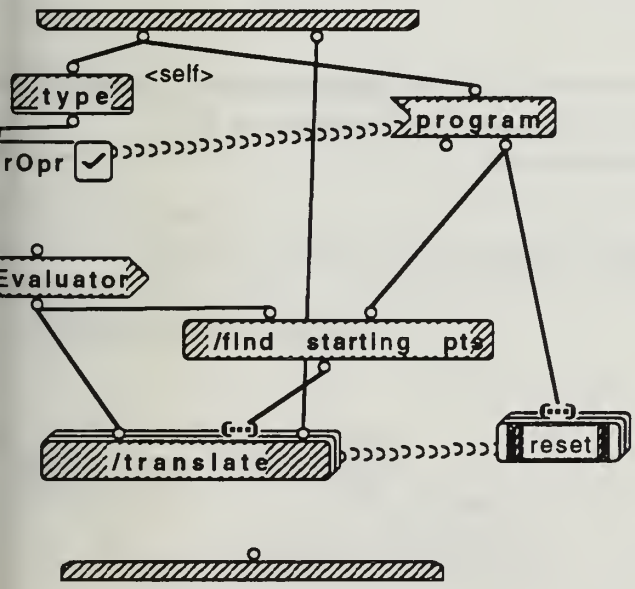



<self>
/terminals
type
DFUsrOpr
( )
connected DFObjects
get opr name
/translate
no operation is attached yet so compute operation for this operator by recursively calling translate to the terminals
detach turtle/open window
(length)
do processing
/rootValue
returns operation object

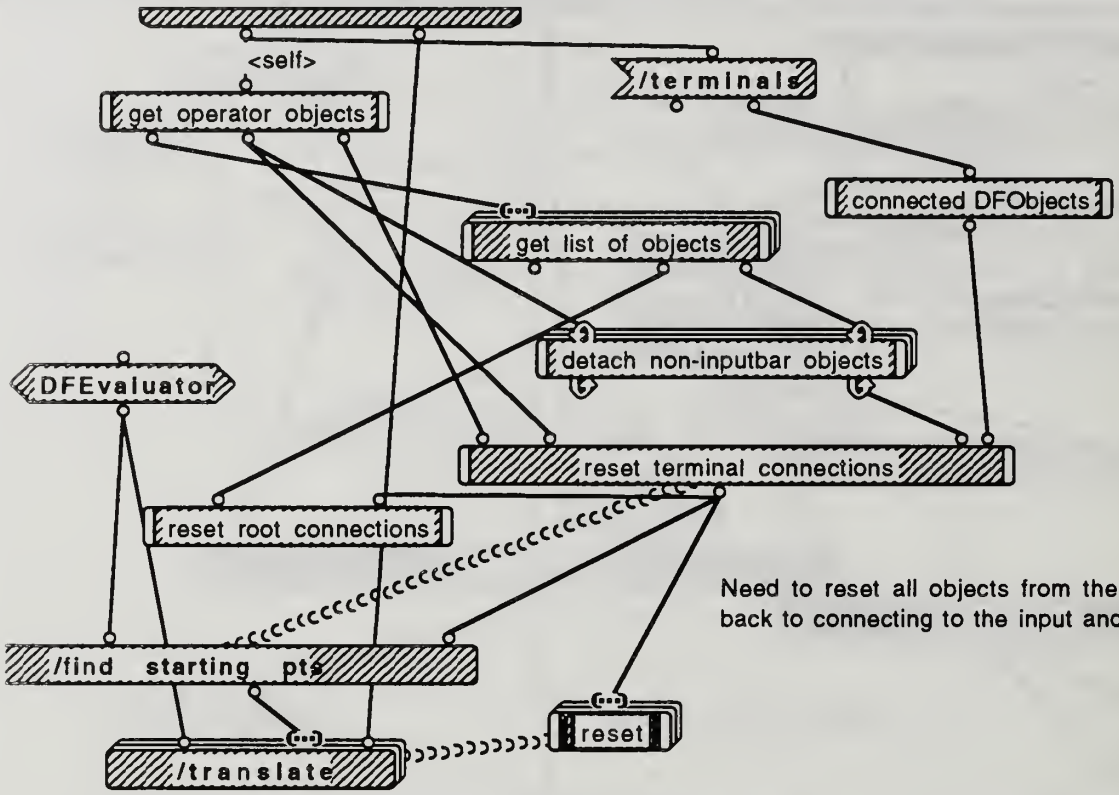
DFOperator/translate 3:5



DFOperator/translate 4:5



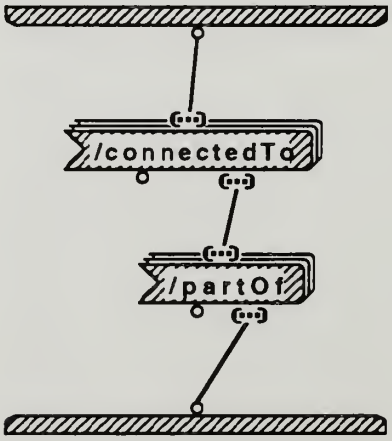
DFOperator/translate 5:5



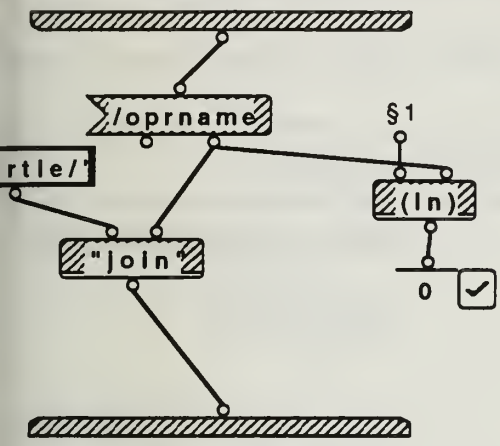
Need to reset all objects from the original definition back to connecting to the input and output objects.

Operator must be a user defined operator. This method needs further testing and modification to allow user to reuse new operator with new input values. Also need to be able to use the turtle coming out of the ne operator.

DFOperator/translate 2:5connected DFObjects 1:1

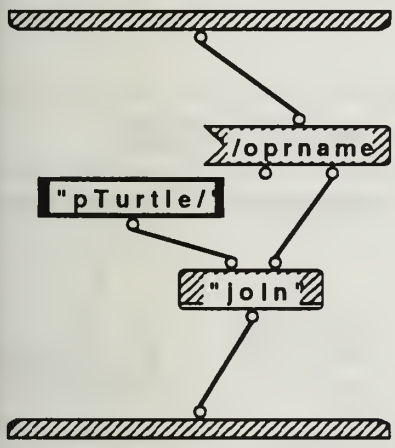


OFOperator/translate 2:5get opr name 1:2

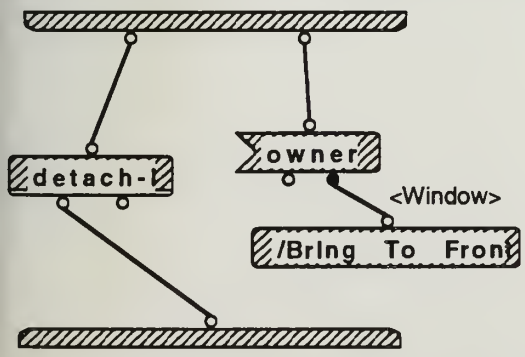


ward" "turnleft" "turnright" "turnto" "goto" "penup" "pendown")

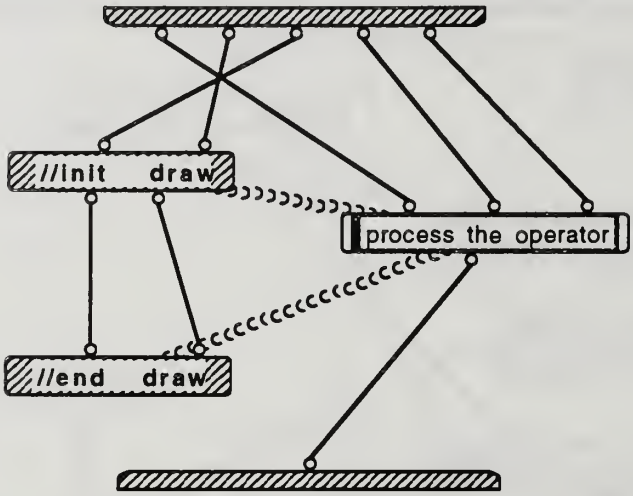
OFOperator/translate 2:5get opr name 2:2



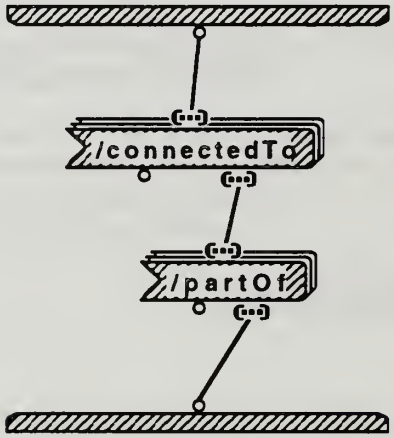
OFOperator/translate 2:5detach turtle/open window 1:1



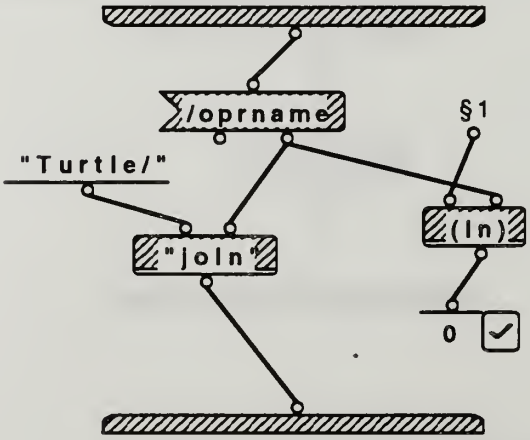
DFOperator/translate 2:5do processing 1:1



DFOperator/translate 3:5connected DFObjects 1:1



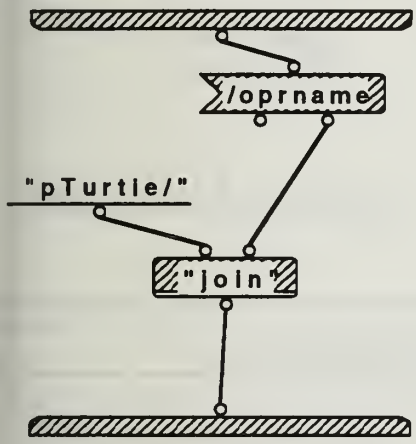
DFOperator/translate 3:5get opr name 1:2



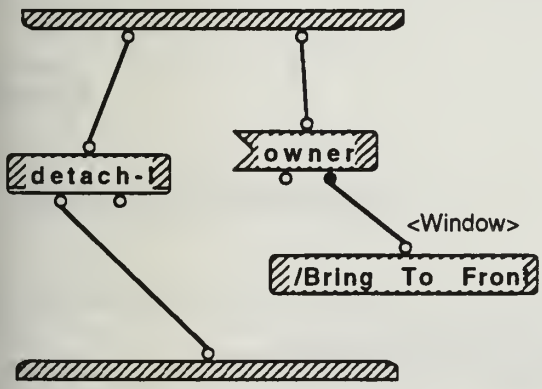
DFOperator/translate 3:5get opr name 1:2

rward" "turnleft" "turnright" "turnto" "goto" "penup" "pendown")

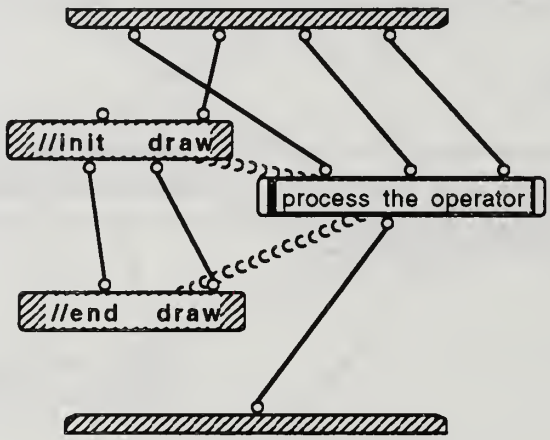
DFOperator/translate 3:5get opr name 2:2



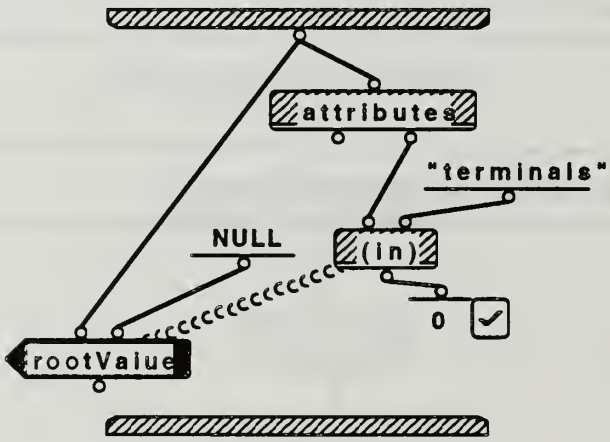
DFOperator/translate 3:5detach turtle/open window 1:1



DFDperator/translate 3:5do processing 1:1

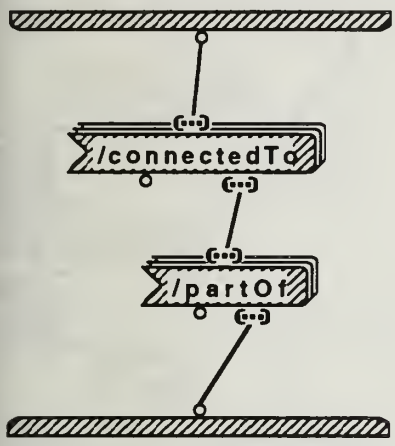


DFOperator/translate 4:5reset 1:2

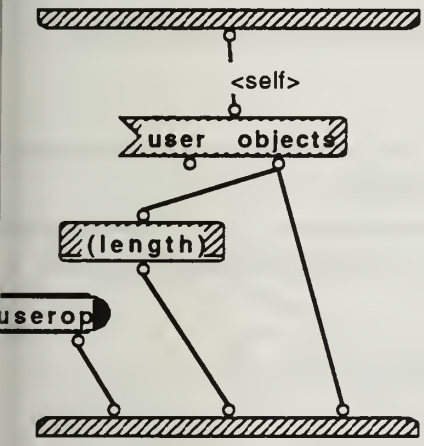


DFDperator/translate 4:5reset 2:2

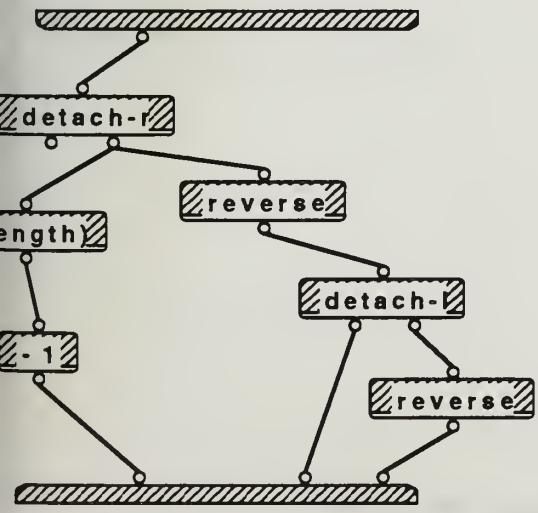
DFOperator/translate 5:5connected OFObjects 1:1



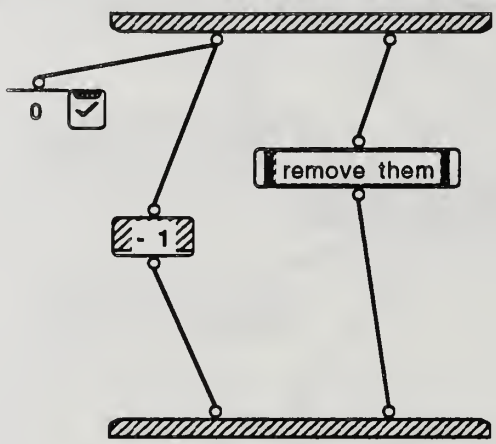
DFOperator/translate 5:5get operator objects 1:1



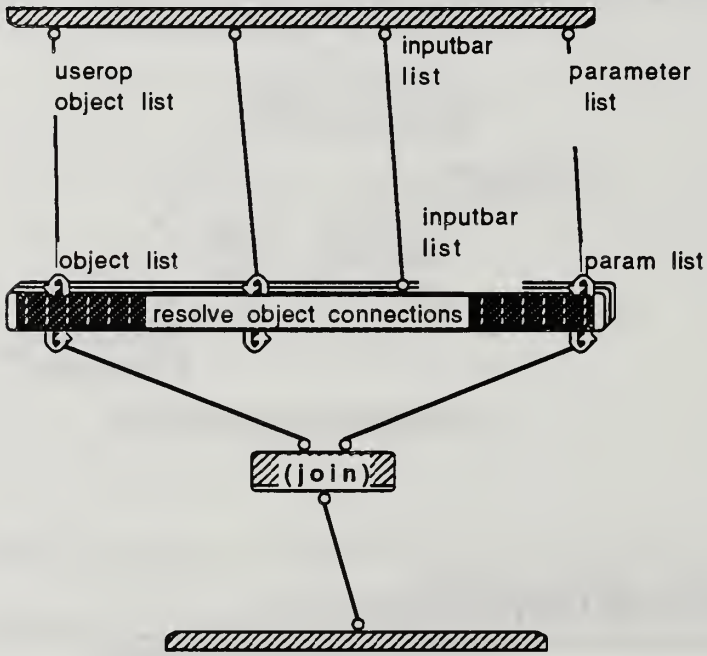
DFOperator/translate 5:5get list of objects 1:1



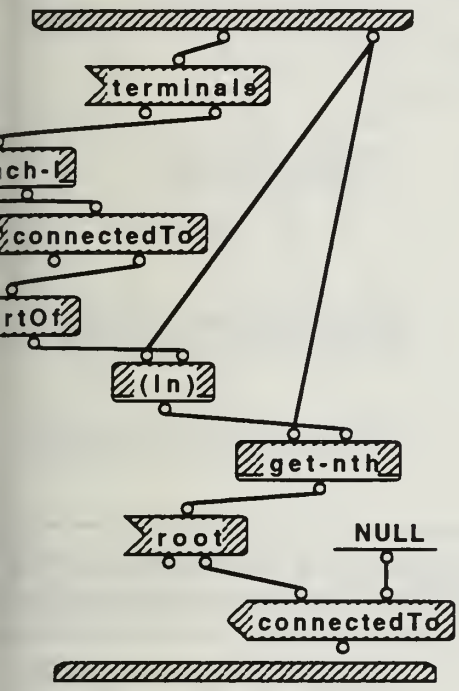
DFOperator/translate 5:5detach non-inputbar objects 1:1



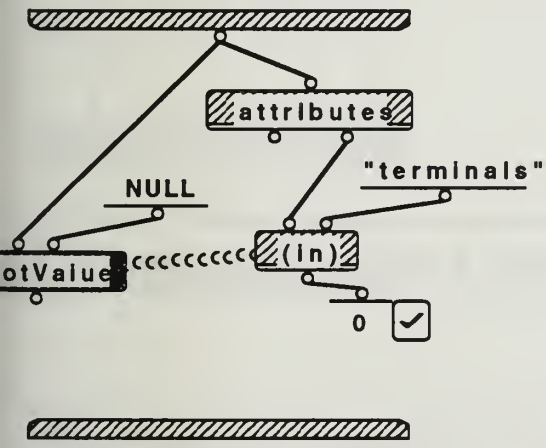
DFOperator/translate 5:5reset terminal connections 1:1



DFOperator/translate 5:5reset root connections 1:1

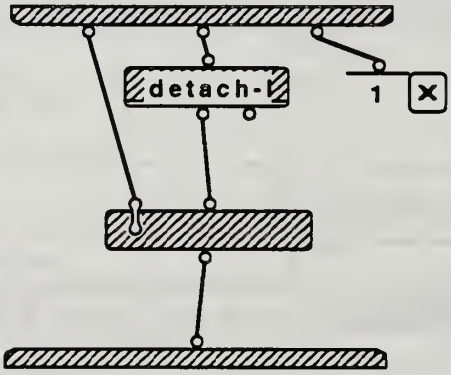


DFOperator/translate 5:5reset 1:2

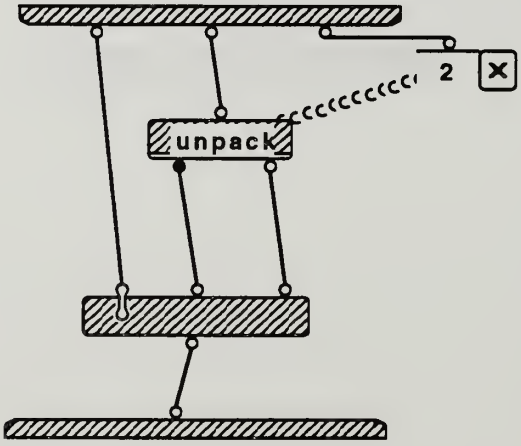


DFDperator/translate 5:5reset 2:2

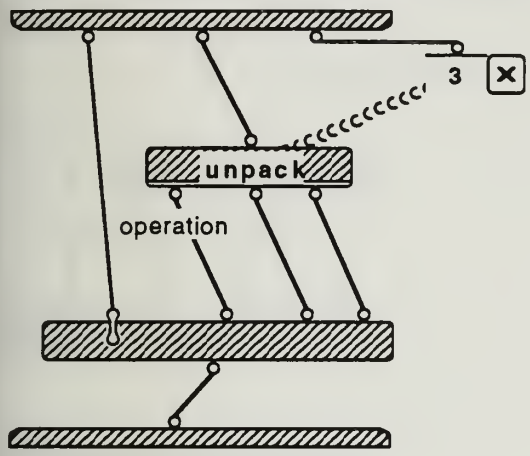
DFDperator/translate 2:5do processing 1:1process the operator 1:4



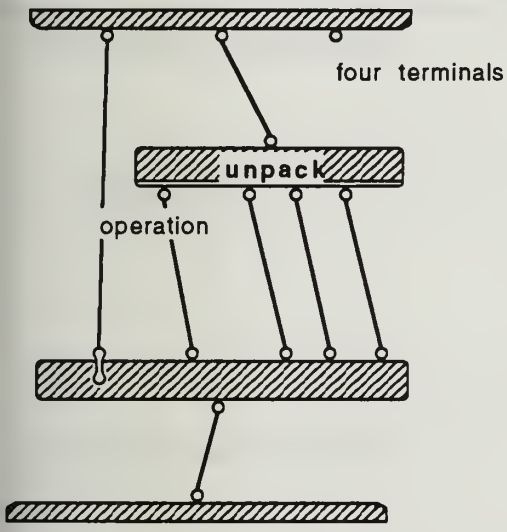
DFOperator/translate 2:5do processing 1:1process the operator 2:4



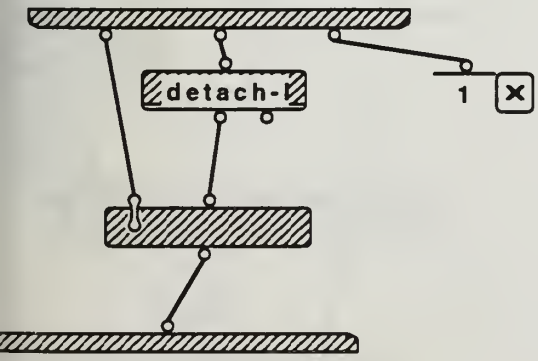
DFOperator/translate 2:5do processing 1:1process the operator 3:4



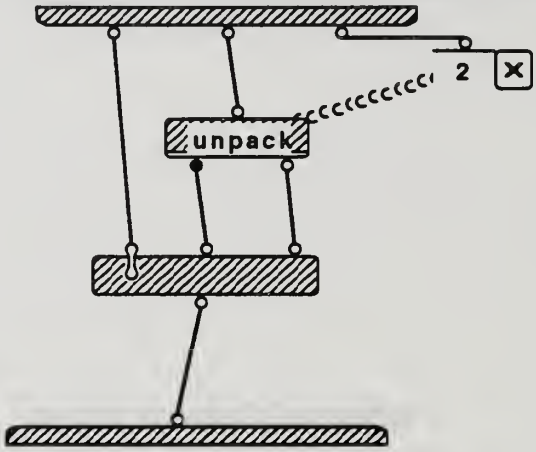
DFOperator/translate 2:5do processing 1:1process the operator 4:4



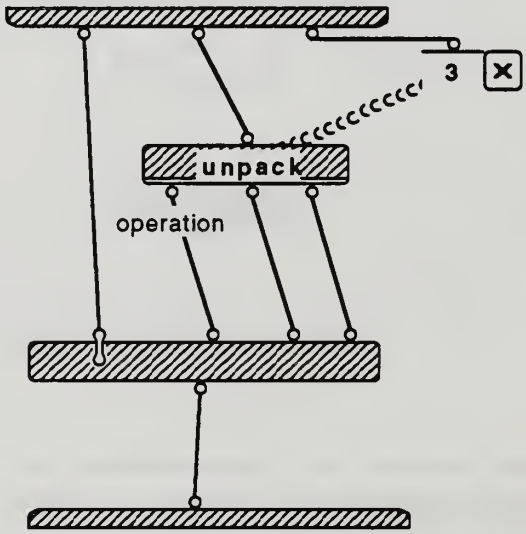
DFOperator/translate 3:5do processing 1:1process the operator 1:4



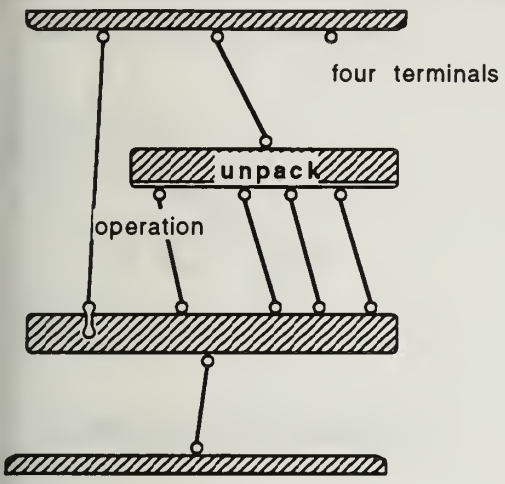
DFOperator/translate 3:5do processing 1:1process the operator 2:4



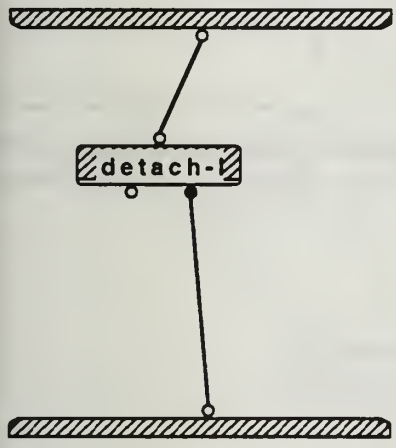
DFOperator/translate 3:5do processing 1:1process the operator 3:4



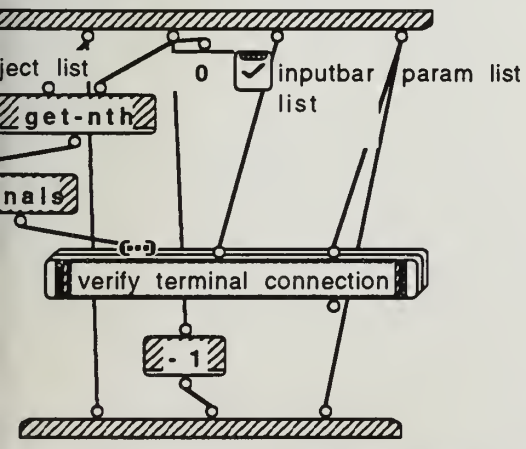
DFDperator/translate 3:5do processing 1:1process the operator 4:4



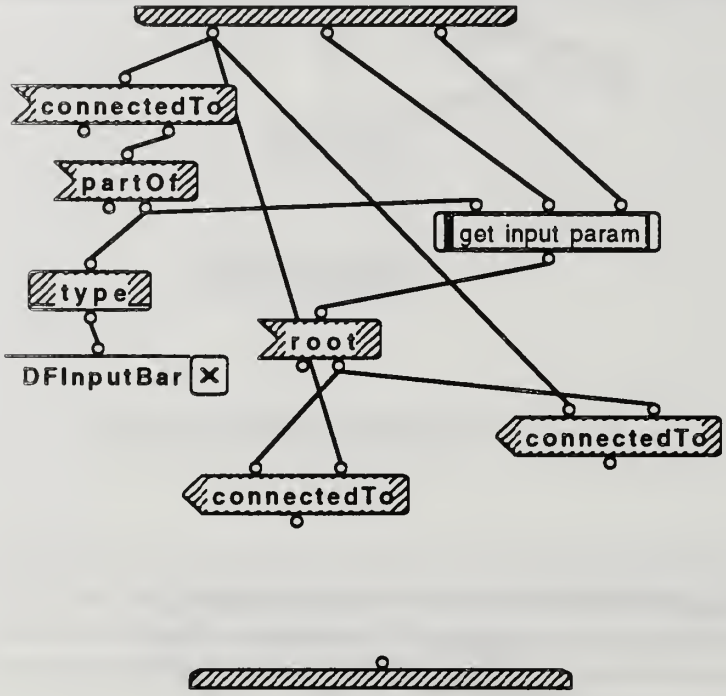
DFDperator/translate 5:5detach non-inputbar objects 1:1remove them 1:1



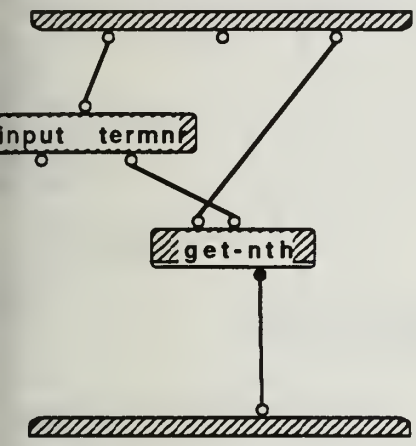
DFOperator/translate 5:5reset terminal connections 1:1resolve object connections 1:1



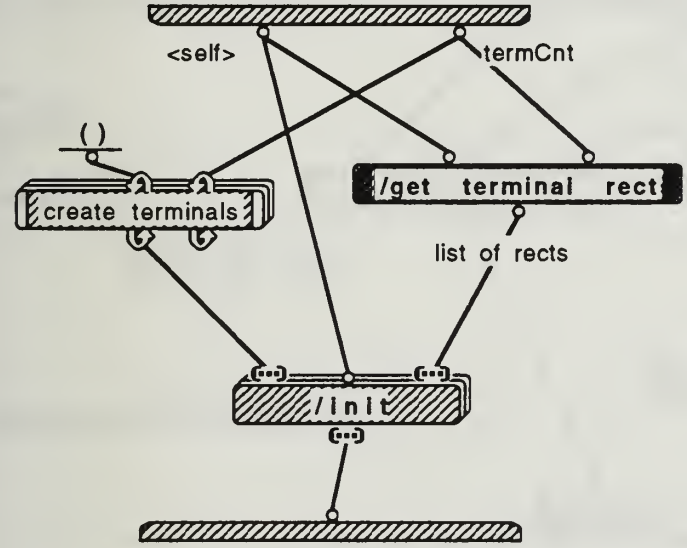
DFOperator/translate 5:5reset terminal connections 1:1resolve object connections 1:1verify terminal connec



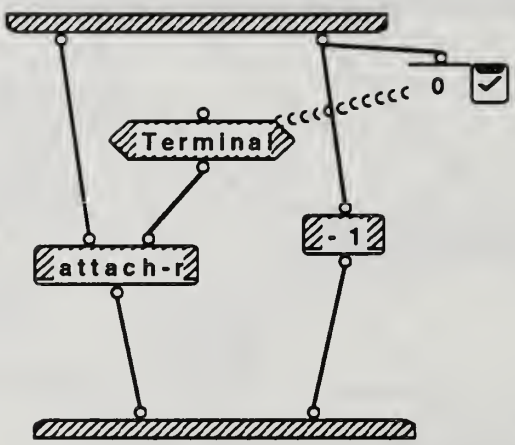
DFOperator/translate 5:5reset terminal connections 1:1resolve object connections 1:1verify terminal connec



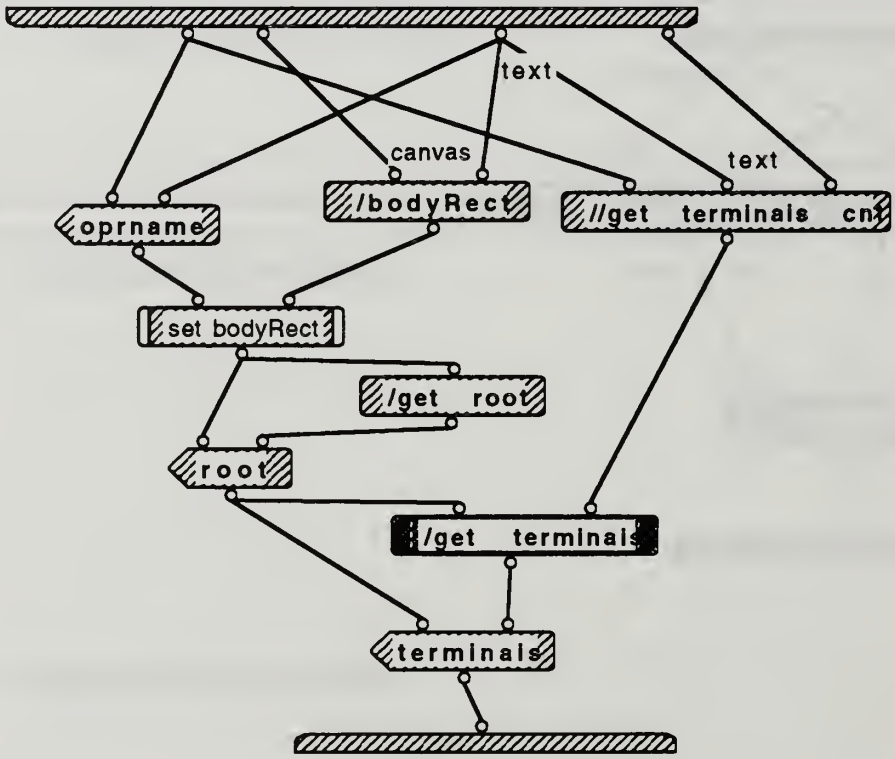
DFOperator/get terminals 1:1



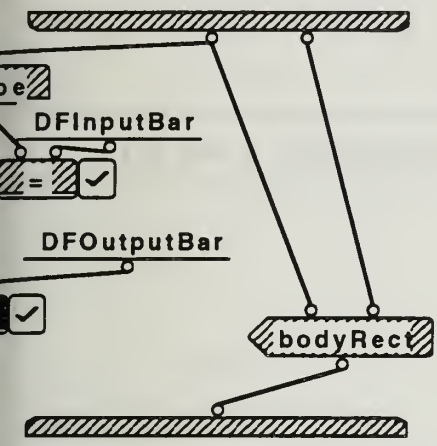
DFOperator/get terminals 1:1create terminals 1:1



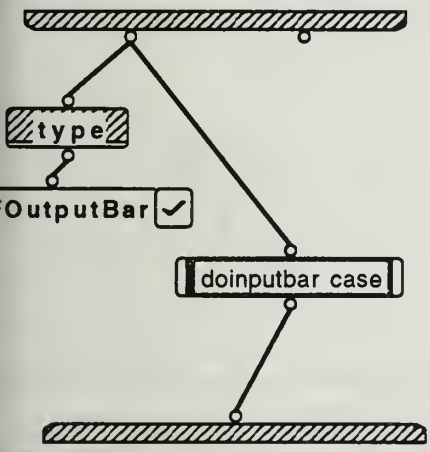
DFOperator/init 1:1



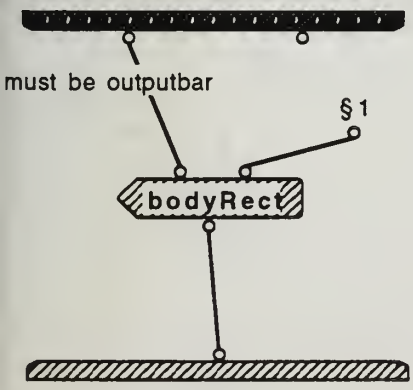
DFOperator/init 1:1set bodyRect 1:3



DFOperator/init 1:1set bodyRect 2:3



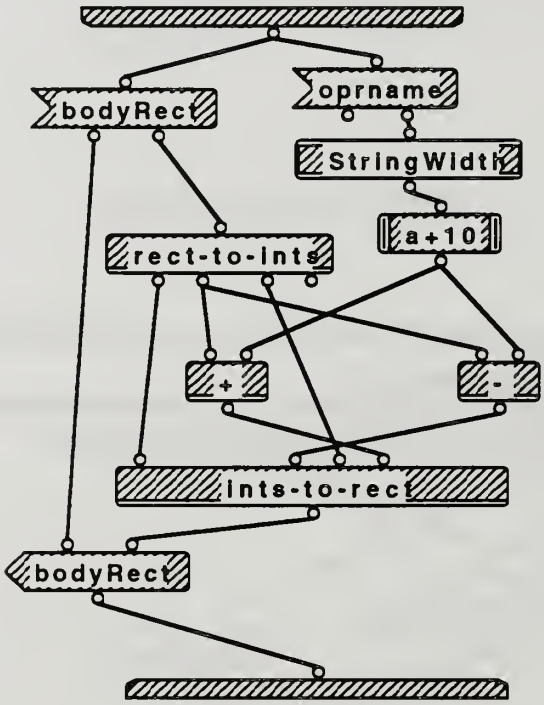
DFOperator/init 1:1set bodyRect 3:3



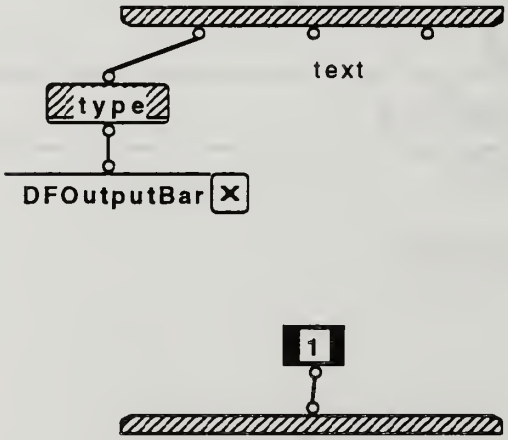
DFOperator/init 1:1set bodyRect 3:3

§1. {200 100 210 400}

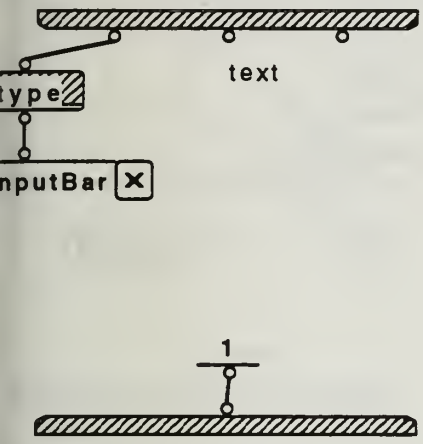
DFOperator/init 1:1set bodyRect 2:3doinputbar case 1:1



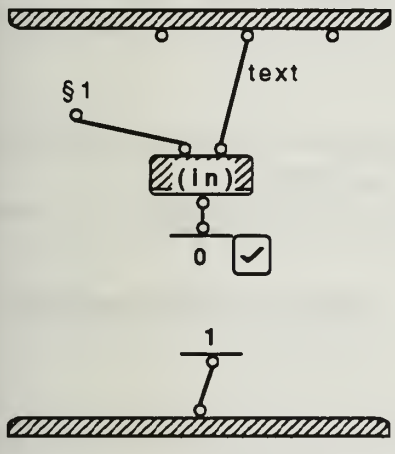
DFOperator/get terminals cnt 1:8



DFOperator/get terminals cnt 2:8

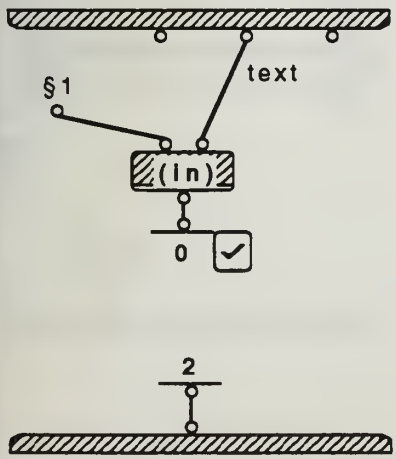


DFOperator/get terminals cnt 3:8



nup" "pendown")

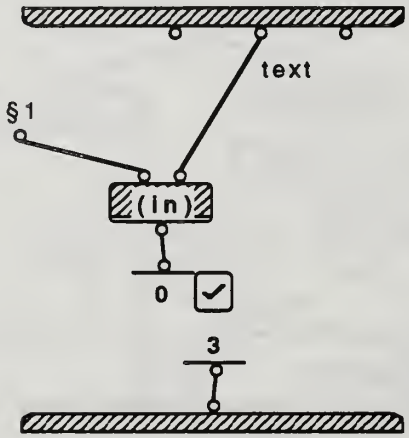
DFOperator/get terminals cnt 4:8



DFOperator/get terminals cnt 4:8

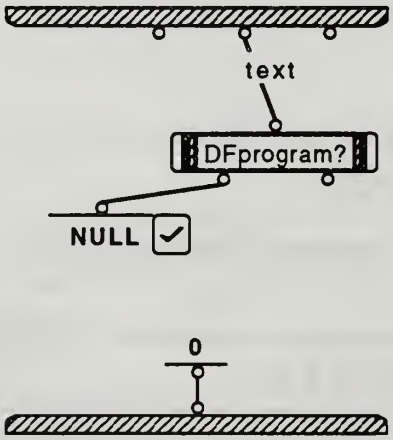
§1. ("forward" "turnright" "turnleft" "turnto" "goto" "drawto")

DFOperator/get terminals cnt 5:8

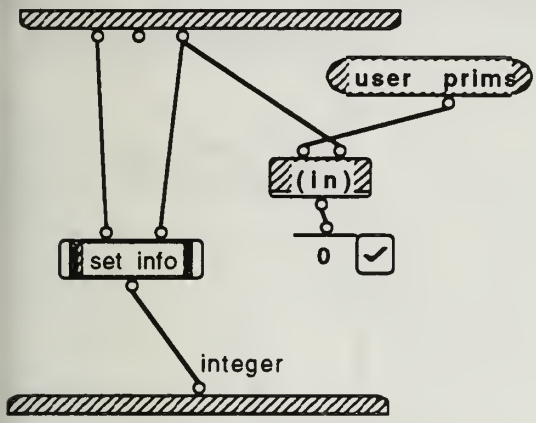


§1. ("square" "triangle" "circle")

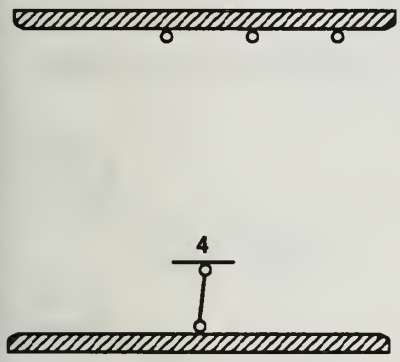
DFOperator/get terminals cnt 6:8



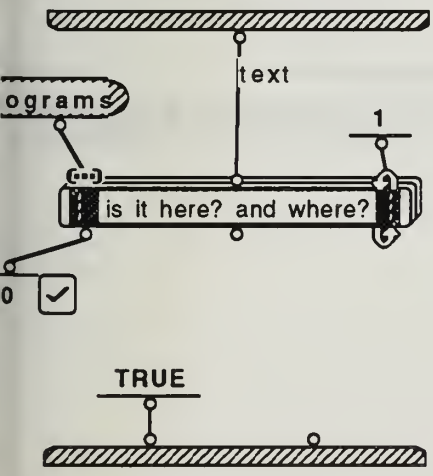
DFOperator/get terminals cnt 7:8



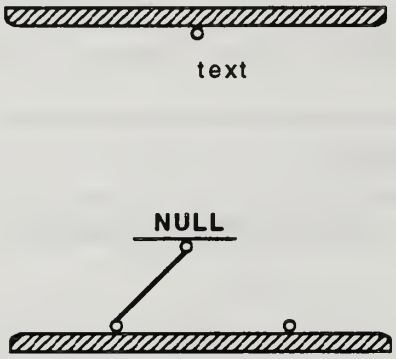
DFOperator/get terminals cnt 8:8



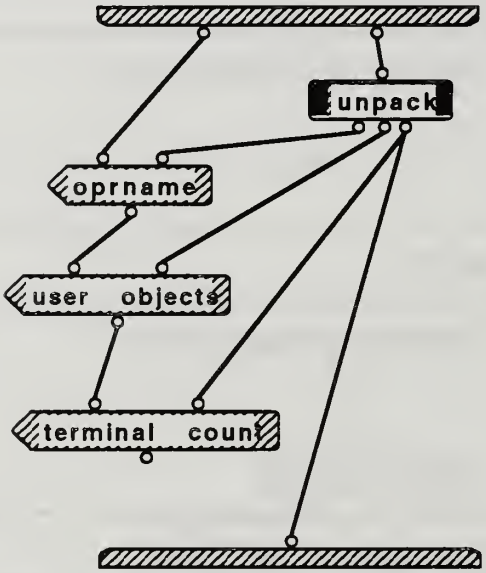
DFOperator/get terminals cnt 6:8DFprogram? 1:2



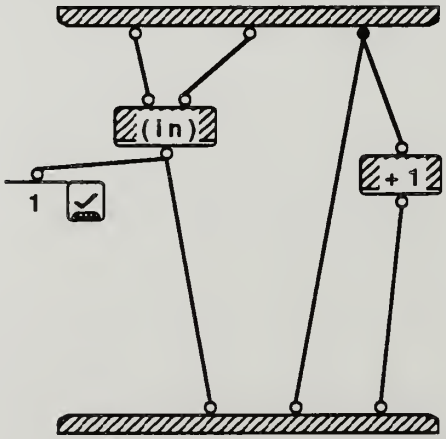
DFOperator/get terminals cnt 6:8DFprogram? 2:2



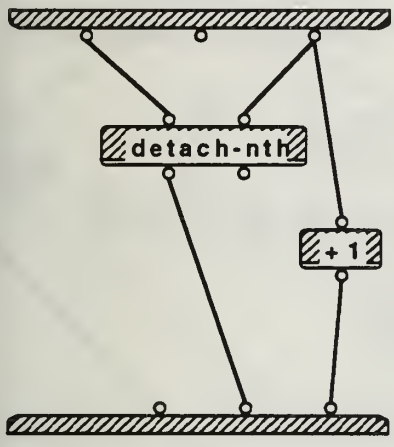
DFOperator/get terminals cnt 7:8set info 1:1



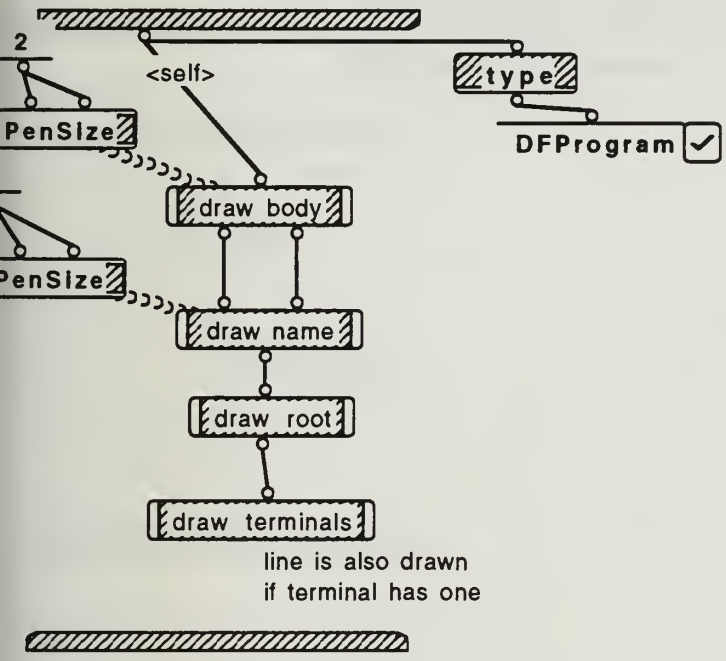
DFOperator/get terminals cnt 6:8DFprogram? 1:2is it here? and where?



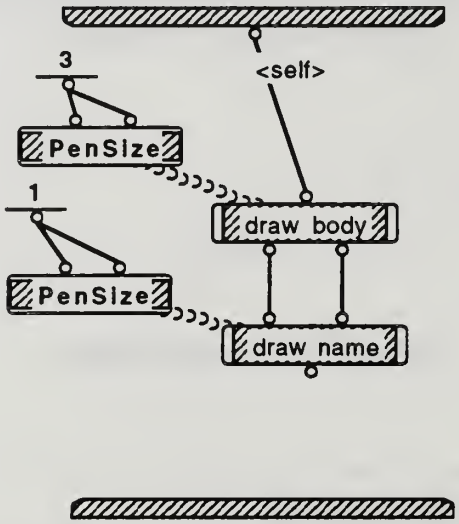
DFOperator/get terminals cnt 6:8DFprogram? 1:2is it here? and where? 2:2



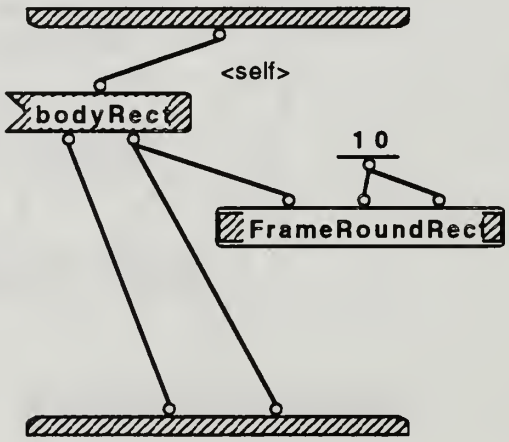
DFOperator/draw 1:2



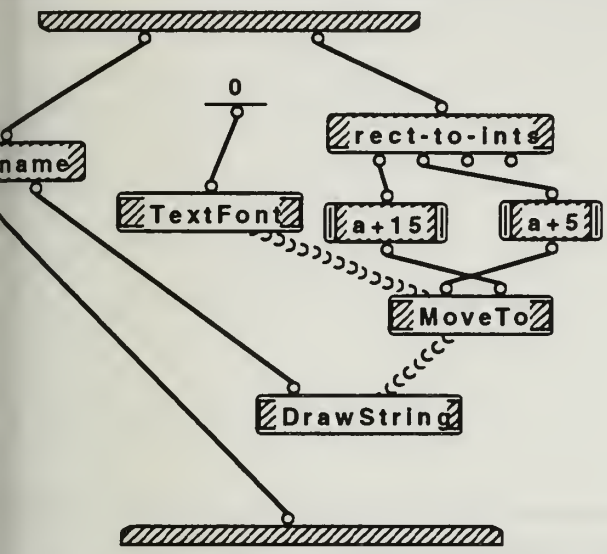
DFOperator/draw 2:2



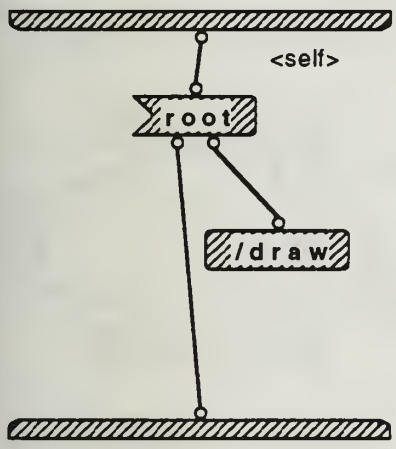
DFOperator/draw 1:2draw body 1:1



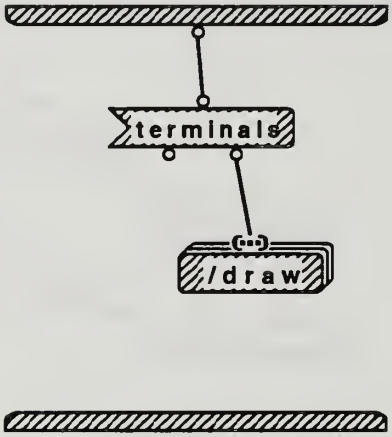
DFOperator/draw 1:2draw name 1:1



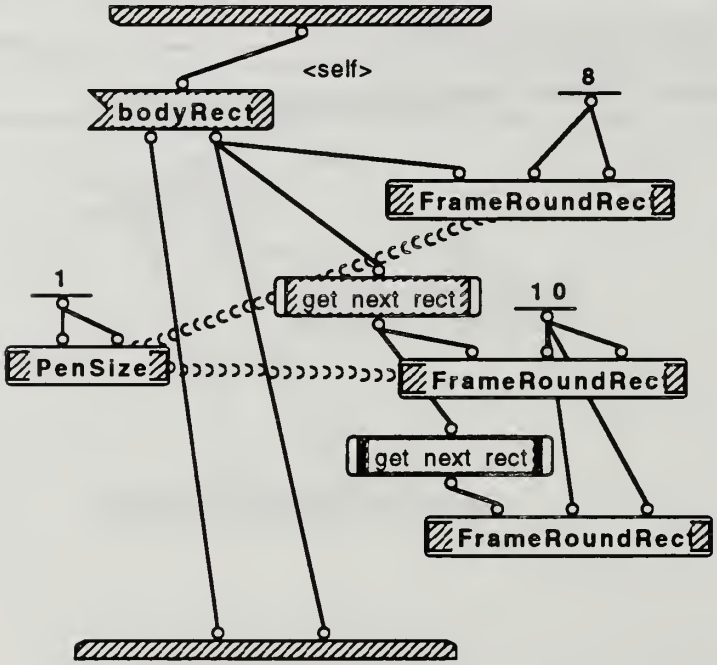
DFOperator/draw 1:2draw root 1:1



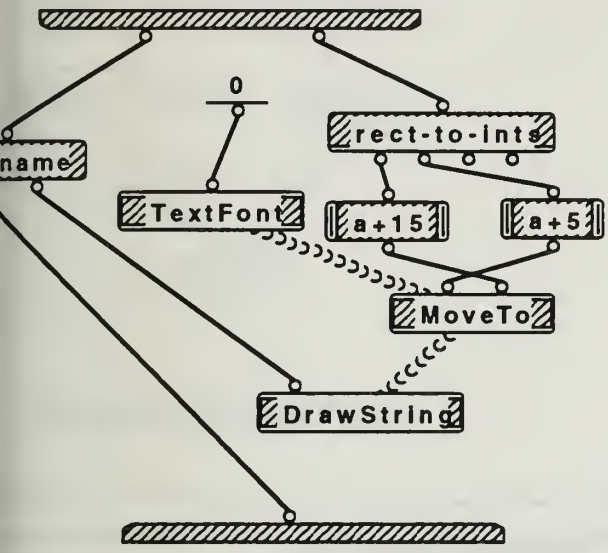
DFOperator/draw 1:2draw terminals 1:1



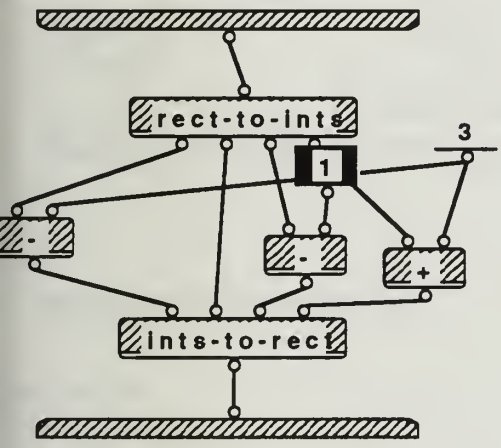
DFOperator/draw 2:2draw body 1:1



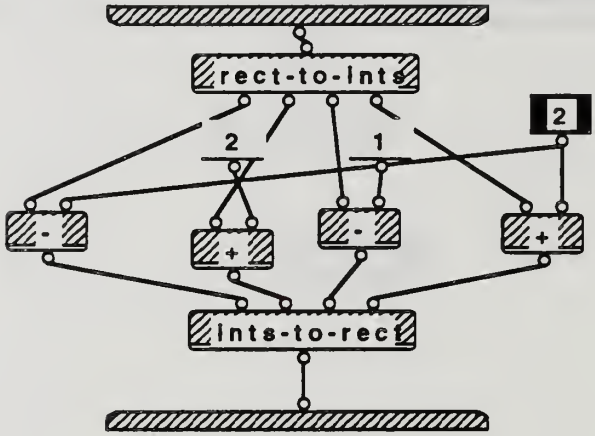
DFDperator/draw 2:2draw name 1:1



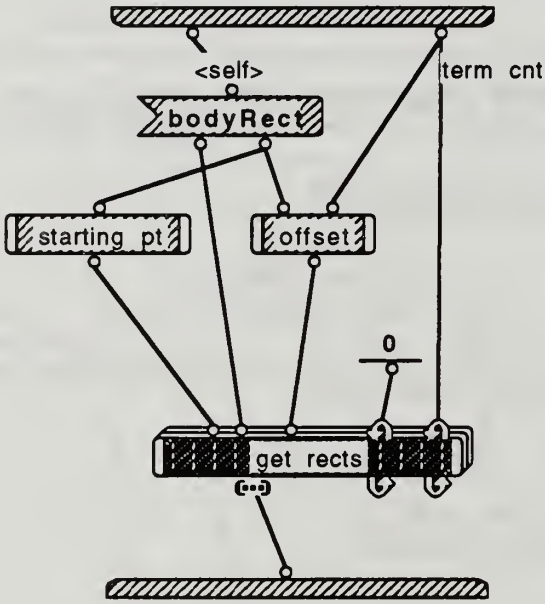
DFDperator/draw 2:2draw body 1:1get next rect 1:1



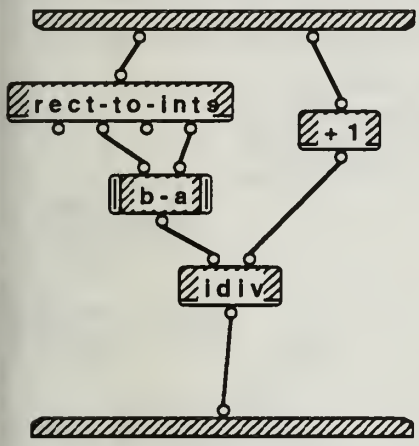
DFOperator/draw 2:2draw body 1:1get next rect 1:1



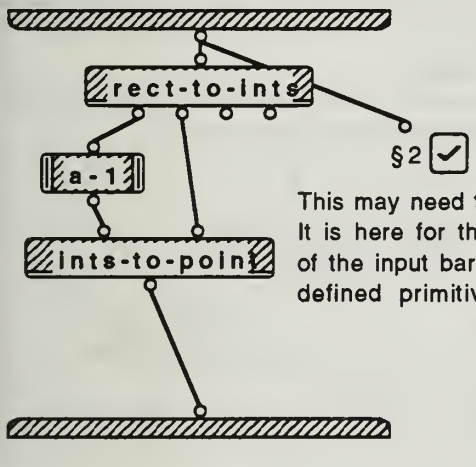
DFOperator/get terminal rects 1:1



DFOperator/get terminal rects 1:1offset 1:1



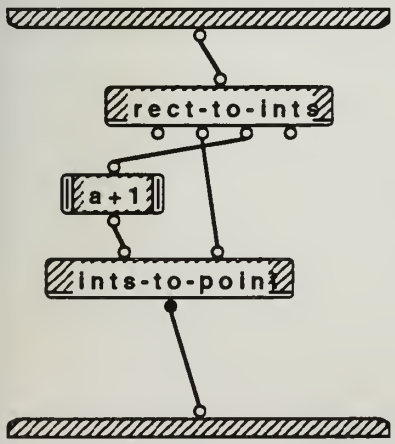
DFOperator/get terminal rects 1:1starting pt 1:2



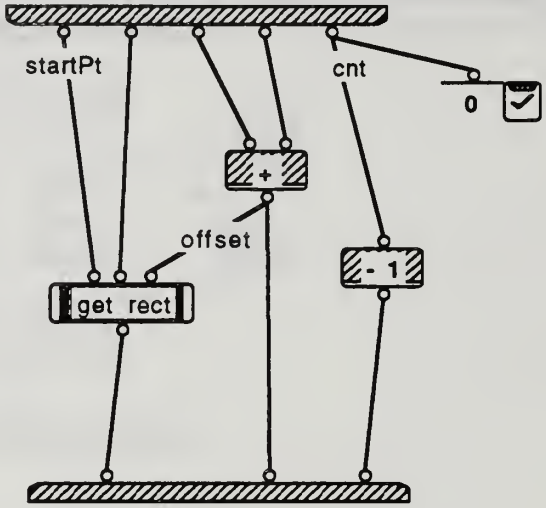
This may need to be removed.
It is here for the implementation
of the input bar for the user
defined primitive.

100 20 400}
100 20 400}

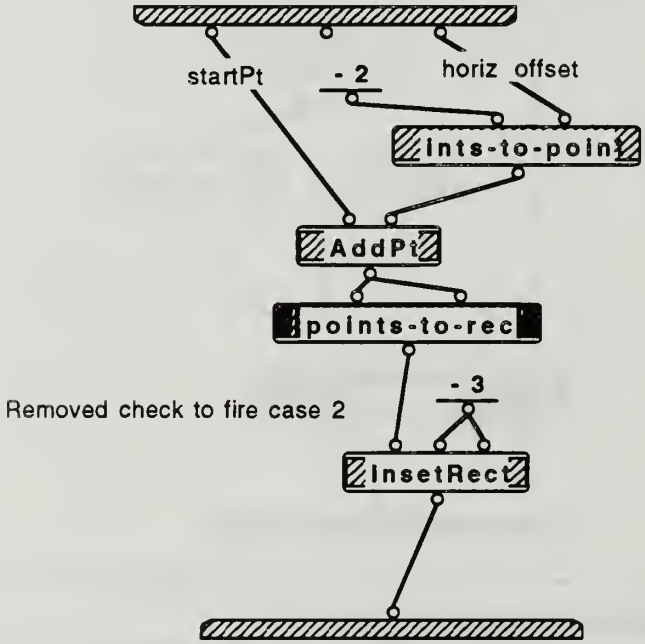
DFOperator/get terminal rects 1:1starting pt 2:2



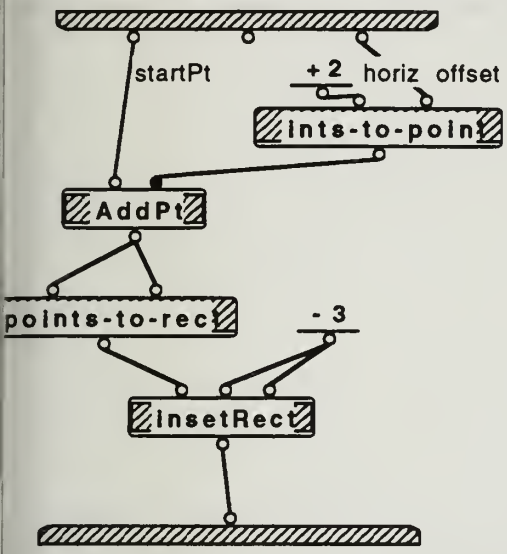
DFOperator/get terminal rects 1:1get rects 1:1



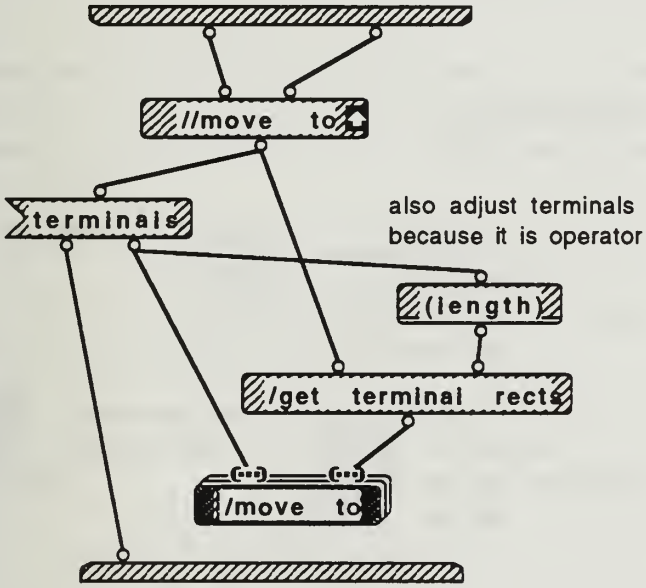
DFOperator/get terminal rects 1:1get rects 1:1get rect 1:2



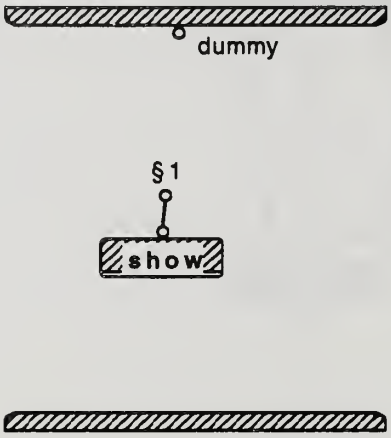
DFOperator/get terminal rects 1:1get rects 1:1get rect 2:2



DFOperator/move to 1:1



DFOperator/remove show info 1:1



§1. show info on operator

DFOperator/end draw 1:1

▽ DFNonOpr

NULL <Root> object

root

{ 0 0 20 0 } location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

textstring

" "

dispstring

DFNonOpr

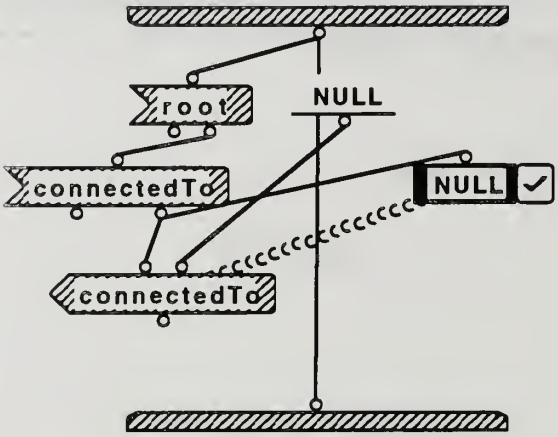
init init itself

terminals returns nothing because DF Text has no terminals; called from terminal click? of process click

disconnect Disconnect from other objects.

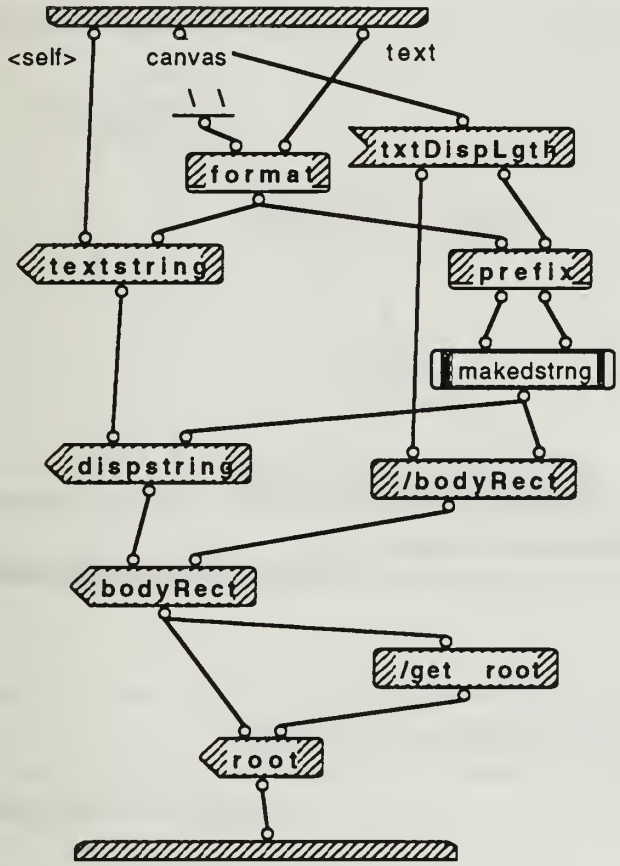
draw draw itself on the "sc-begin"ed canvas

DFNonOpr/disconnect 1:2

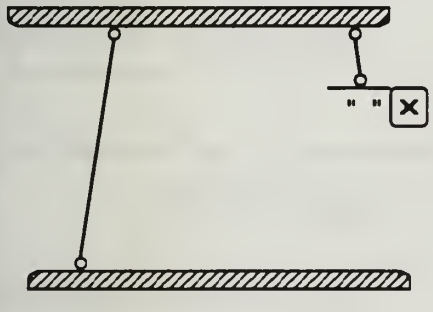


DFNonOpr/disconnect 2:2

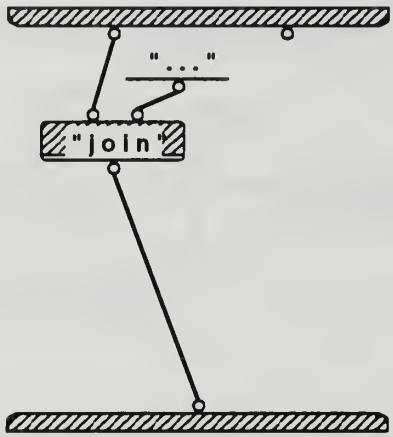
DFNonOpr/init 1:1



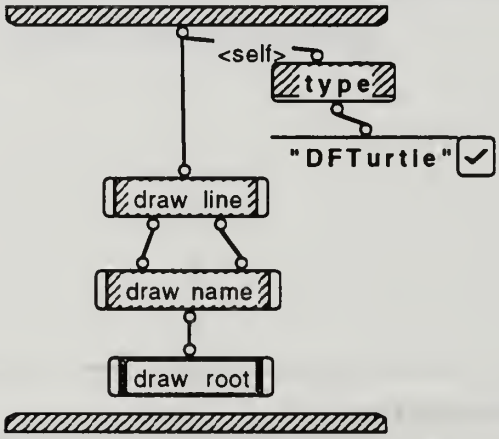
DFNonOpr/init 1:1makedstrng 1:2



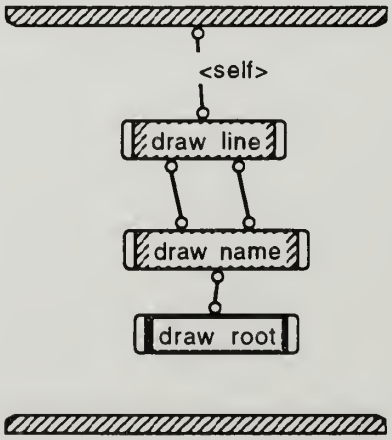
DFNonOpr/init 1:1makedstrng 2:2



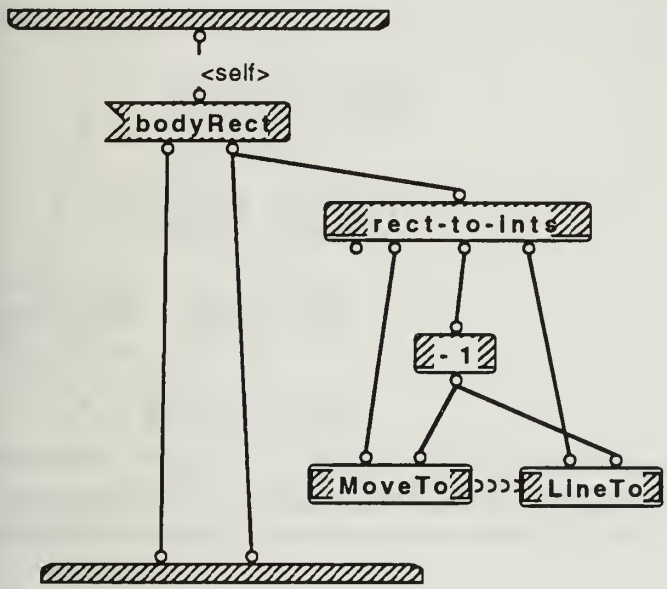
DFNonOpr/draw 1:2



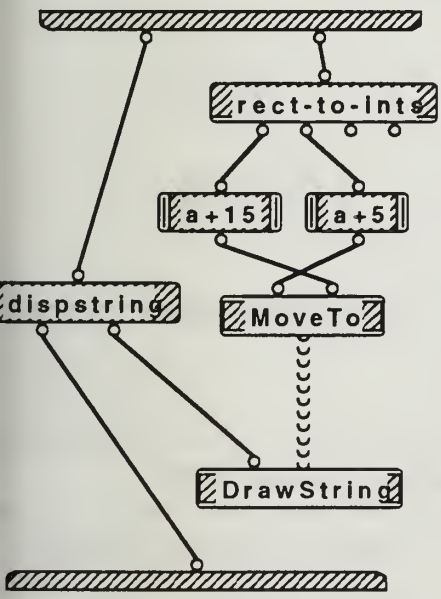
DFNonOpr/draw 2:2



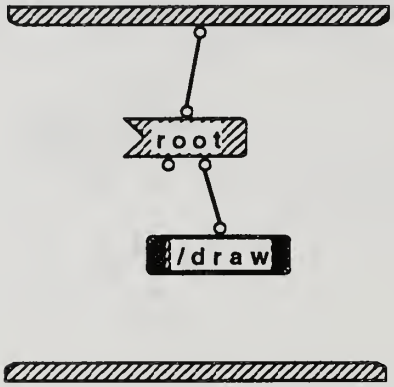
DFNonOpr/draw 1:2draw line 1:1



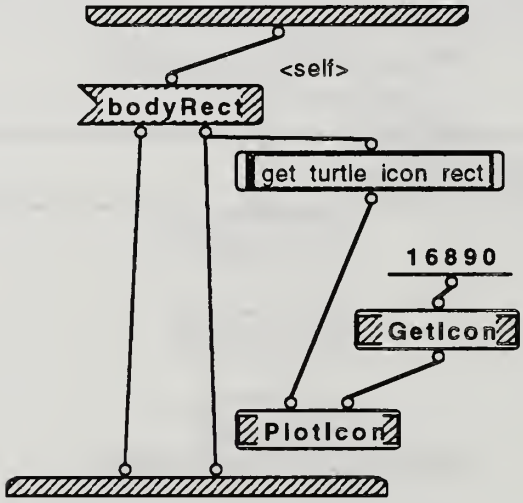
DFNonOpr/draw 1:2draw name 1:1



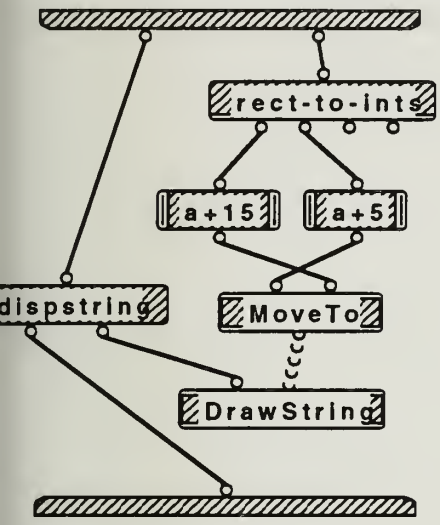
DFNonOpr/draw 1:2draw root 1:1



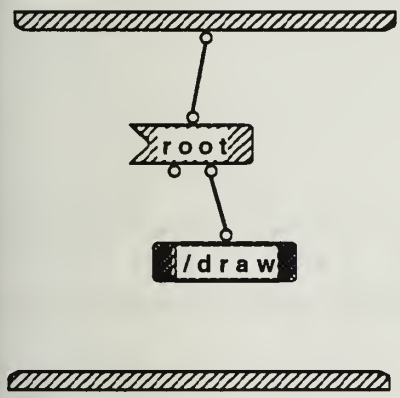
DFNonOpr/draw 2:2draw line 1:1



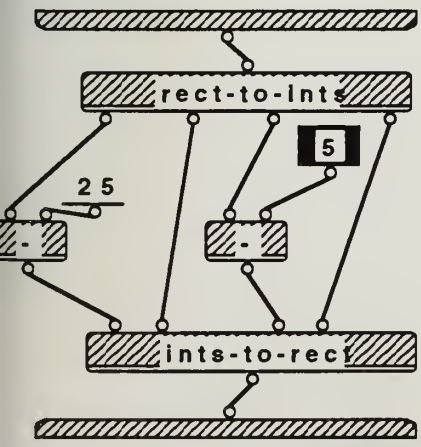
DFNonOpr/draw 2:2draw name 1:1



DFNonOpr/draw 2:2draw root 1:1



DFNonDpr/draw 2:2draw line 1:1get turtle icon rect 1:1



▽ DFTurtle

NULL <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

textstring

" "

dispstring

NULL

name

DFTurtle

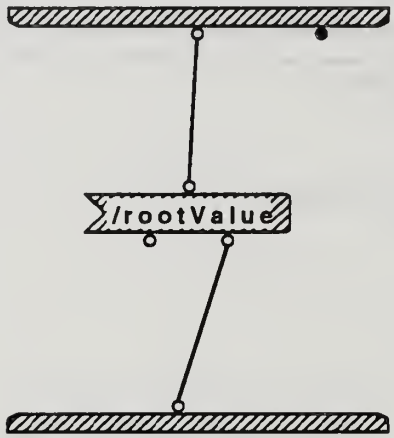
returns turtle object

ranslate

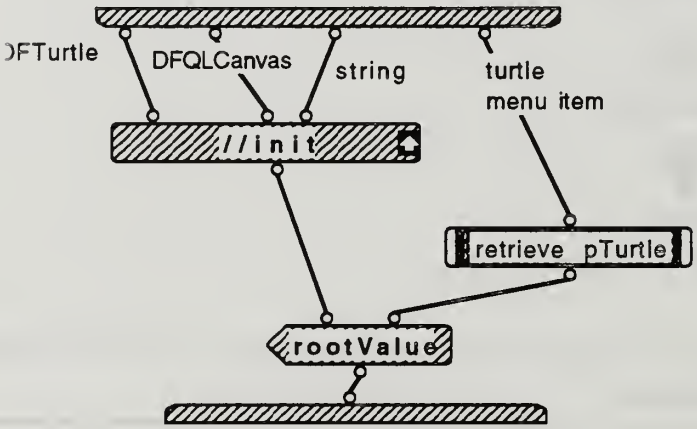
init

Initialize DFTurtle object.

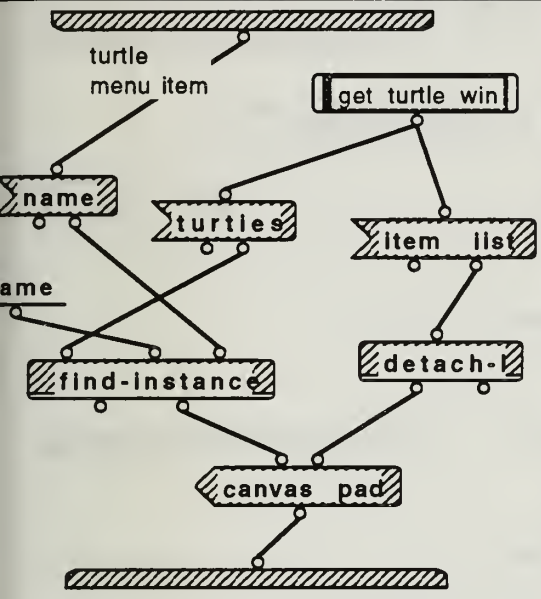
DFTurtle/translate 1:1



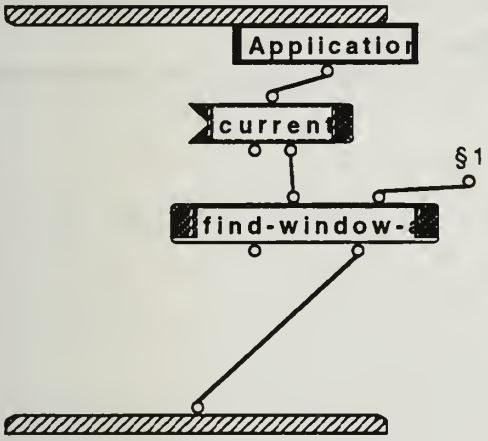
DFTurtle/init 1:1



DFTurtle/init 1:1retrieve pTurtle 1:1



DFTurtle/init 1:1retrieve pTurtle 1:1get turtle win 1:1



tle Display

▽ DFParameter

NULL <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

textstring

" "

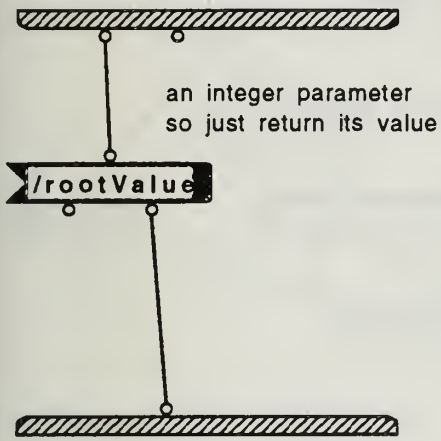
dispstring

DFParameter

translate

returns the integer value

DFParameter/translate 1:1



▽ DFPrimOpr

{ 20 0 28 0 <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

oprname

NULL list of <Terminal>
--((termrect fromObjInstnum) ..)

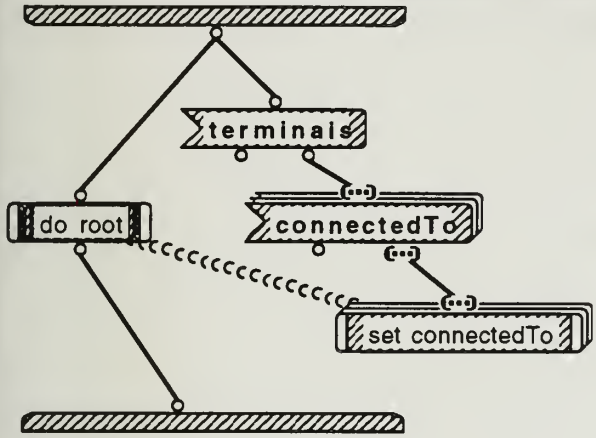
terminals

DFPrimOpr

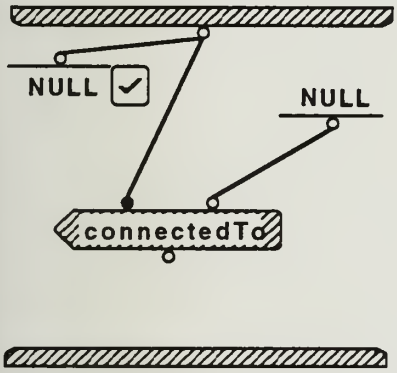
disconnec

input: DFOperator
output: DFOperator
This disconnects the object from others.

DFPrimOpr/disconnect 1:1

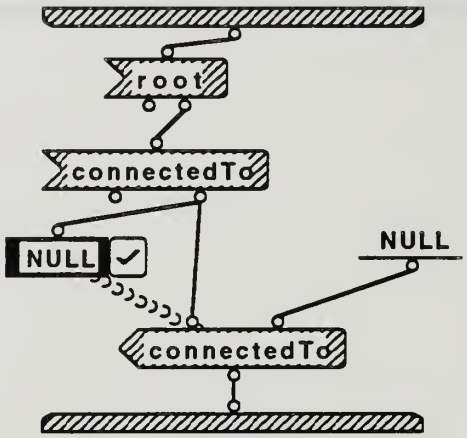


DFPrimOpr/disconnect 1:1set connectedTo 1:2

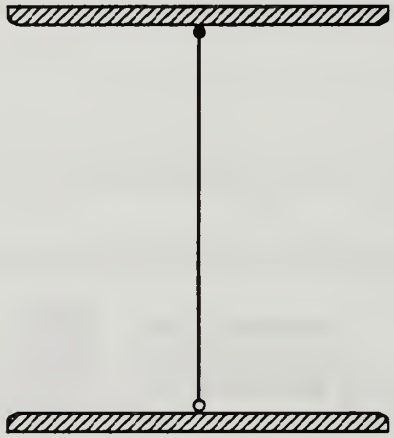


DFPrimOpr/disconnect 1:1set connectedTo 2:2

DFPrimOpr/disconnect 1:1do root 1:2



DFPrimOpr/disconnect 1:1do root 2:2



▽ DFUsrOpr

{ 20 0 28 0 <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

oprname

NULL list of <Terminal>
--((termrect fromObjInstnum) ..)

terminals

NULL

terminal count

NULL

user objects

▽ DFInputBar

NULL <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

oprname

( ) list of <Terminal>
--((termrect fromObjInstnum) ..)

terminals

NULL

input termnr

▽ DFOutputBar

NULL <Root> object

root

{ 0 0 20 0 location of its body

bodyRect

NULL

rootValue

FALSE

selected?

" "

oprname

( ) list of <Terminal>
--((termrect fromObjInstnum) ..)

terminals

NULL

output terminals

▽ DFEvaluator

DFEvaluator

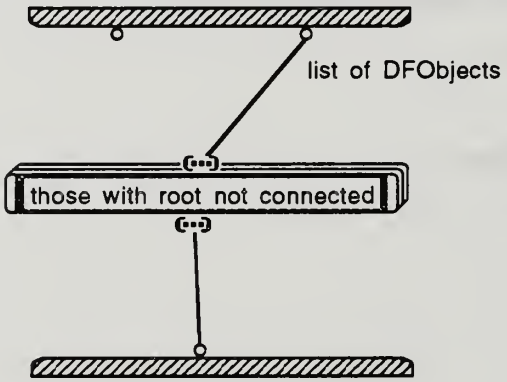
find starting pts

given a list of DFObjects, it returns those with roots not connected, i.e. starting pts for program execution

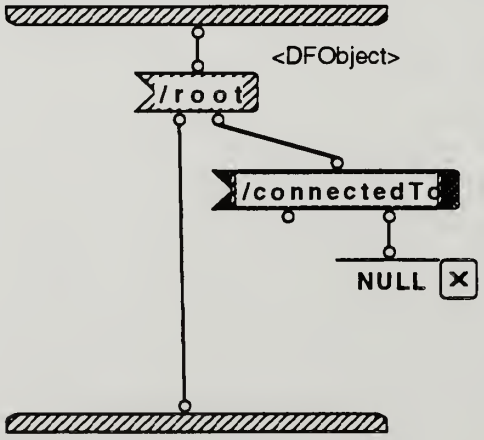
translate

translate/execute program

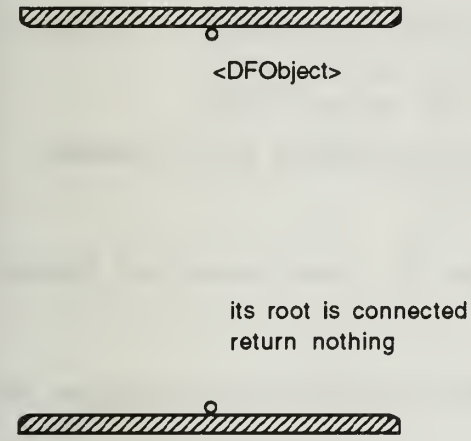
DFEvaluator/find starting pts 1:1



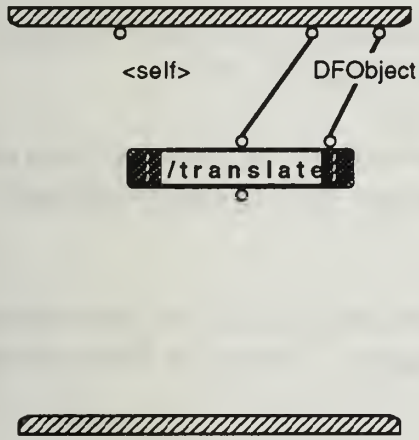
DFEvaluator/find starting pts 1:1those with root not connected 1:2



DFEvaluator/find starting pts 1:1those with root not connected 2:2



DFEvaluator/translate 1:1



LIST OF REFERENCES

[Booc91] Booch, G., *Object-Oriented Design with Applications*, The Benjamin/ Cummings Publishing Company, Inc., 1991.

[CIL86] Chang, S., Ichikawa, T., and Ligomenides, P., *Visual Languages*, Plenum Press, New York and London, 1986.

[Clay88] Clayson, J., *Visual Modeling with LOGO*, The MIT Press Cambridge, MA. 1988.

[CN88] Clements, D., and Natasi, B., "Social and cognitive interactions in educational computer environments." *American Educational Research Association Journal*, 1988.

[Fraz87] Frazier, M., "The effects of Logo on angle estimation skills on 7th graders." Unpublished Master's thesis, Wichita State University, 1987.

[Hare88] Harel, I., "Software design for learning mathematics: on learning Logo and fractions through instructional software design." MIT Epistemology and Learning Center, Cambridge, MA. 1988.

[LGL88] Lehrer, R., Guckenberg, T., and Lee, O.," Comparative study of the cognitive consequences of inquiry-based Logo instruction." *Journal of Educational Psychology*, 1988.

[Pape80] Papert, S., *Mindstorms; children, computers, and powerful ideas*, Basic Books, New York, 1980.

[Shu88] Shu, N., *Visual Programming*, Van Nostrand Reinhold Company, 1988.

[TGS88a] The Gunakara Sun Systems, *Prograph Tutorial*, 1988.

[TGS88b] The Gunakara Sun Systems, *Prograph Reference*,1988.

[TGS91] The Gunakara Sun Systems, *Prograph 2.5 Updates,* 1991.

[YM90] Yoder, S., and Moursund, D., *Logo PLUS for Educators: A Problem Solving Approach and LogoWriter for Educators: A Problem Solving Approach*. ISTE, Eugene, OR, 1990.

BIBLIOGRAPHY

[Bran87] Brand, S., *The Media Lab: Inventing the Future at MIT*, Viking Penguin Inc., 1987.

[Chan90] Chang, S., *Principles of Visual Languages*, Prentice-Hall Inc., Englewood Cliffs, NJ. 1990

[GF87] Goldenberg, P., and Feureig, W., *Exploring Language with Logo*, The MIT Press, Cambridge, MA. 1987.

[KL89] Kim, W., and Lochovsky, F., *Object-Oriented Concepts, Databases, and Applications*, ACM PRESS, New York, New York, 1989.

[Laur90] Laurel, B., *The Art of Human Computer Interface Design*, Addison-Wesley Publishing, 1990.

[Logo80] Logo Computer Systems Inc., *Guide To Programming*, 1980.

[OS83] O'Shea, T., and Self, J., *Learning and Teaching with Computers*, Prentice-Hall Inc., Englewood Cliffs, NJ. 1983.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
4.	C. Thomas Wu, Code CS/Wu Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
5.	David A. Erickson, Code CS/Er Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
6.	John Daley, LCDR, USN, Code CS/Da Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	1
7.	Robert S. Lovejoy, LT/USN 9424 S. 83rd Ave. Hickory Hills, Il 60457	2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

GAYLORD S

3 2768 00308754 5